VAX–11
SORT/MERGE
User's Guide

Order No. AA–D113C–TE

May 1982

This manual describes how to use the VAX–11 native mode SORT/MERGE utility. The manual is intended for all users.

OPERATING SYSTEM AND VERSION: VAX/VMS V3

SOFTWARE VERSION: VAX/VMS V3

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license, and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	VT	IAS
DECUS	DECsystem–10	MASSBUS
DECnet	DECSYSTEM 20	PDT
PDP	DECwriter	RSTS
UNIBUS	DIBOL	RSX
VAX	Edusystem	VMS
		digital

Commercial Engineering Publications typeset this manual using DIGITAL's TMS–11 Text Management System.

Contents

Appendix D Data Types

Glossary

Index

Figures

Tables

Preface

Document Objectives

The *VAX–11 SORT/MERGE User's Guide* describes how to use the SORT/MERGE utility of the VAX/VMS Operating System, Version 3.

Intended Audience

This manual is intended for users familiar with VAX/VMS. VAX–11 SORT/MERGE takes two forms: a utility invoked by a command line and a package of subroutines callable from a user program. Chapters 1 and 2, which describe the utility, are written so that users with little or no programming experience can understand how to use the SORT/MERGE utility. Chapter 3, which describes the callable subroutines, is written for programmers experienced with one or more VAX–11 native mode languages.

Document Structure

Chapter 1 introduces the SORT/MERGE utility and shows how to execute a simple sort or merge operation.

Chapter 2 describes all qualifiers, subqualifiers, and parameters that can be used with the SORT/MERGE utility.

Chapter 3 explains how to call SORT and MERGE subroutines from a user program.

Chapter 4 provides ways to improve SORT efficiency.

Appendix A lists SORT/MERGE error messages with recovery procedures.

Appendixes B, C, and D contain the following programming aids: octal/hexadecimal/decimal conversion charts, ASCII character set, and data types used by VAX–11 SORT/MERGE.

The Glossary defines terms used in the manual.

Finally, page references to key terms appear in the index.

Associated Documents

Familiarity with the following documents is helpful:

VAX/VMS Primer

VAX/VMS Command Language User's Guide

Introduction to VAX–11 Record Management Services

For programmers who use a VAX–11 native mode language, such as VAX–11 FORTRAN, the related language manual provides necessary information.

For a complete list of all VAX–11 documents, including brief descriptions of each, see the *VAX–11 Information Directory*.

Conventions

This manual uses the following conventions:

$	The dollar sign is the system prompt. It indicates that the VAX/VMS command interpreter is ready for command input. Also, the $ must appear in the first character position of a command in an indirect command procedure.
RET	The return represents a carriage return/line feed. Pressing this key after entering a full command line ends the command input and begins processing.
[ ]	Square brackets indicate the enclosed portion is optional.
{ }	Braces indicate options where you must select one in the vertical list.
n	The n indicates variable data input (typically some number value).
_	Underscore indicates an entered underscore character.
^	The circumflex represents a control character. For example, ^C represents pressing the keyboard "Control" character and the letter C at the same time.
red print	Indicates characters you type at the terminal. All characters the system prints or displays are in black.

Chapter 1
Getting Started

Most organizations keep records. As changes occur, they must rearrange these records or combine several information files. Organizations as different as banks, research laboratories, schools, and hospitals share the need for a fast and simple means of arranging their records and printing, processing, or storing the reordered information. VAX–11 SORT/MERGE provides a fast and easy way to sort and merge records stored on any VAX/VMS device.

This chapter:

- Introduces the VAX–11 SORT and MERGE utilities
- Shows the syntax required for simple sort and merge operations and gives examples
- Mentions circumstances in which you need additional qualifiers
- Describes the procedure for running batch SORT and MERGE

You can run SORT or MERGE interactively at a terminal, as a batch job, or as part of a program you write. Section 1.1 discusses executing a simple sort at a terminal; Section 1.2 does the same for a merge. Batch jobs are described in Section 1.3. Chapter 2 gives a full explanation of interactive sorting and merging and describes all qualifiers. Chapter 3 explains how to call SORT or MERGE from a user program.

1.1 Sorting Records

VAX–11 SORT takes records from one or more input files (up to 10), sorts them according to the key field(s) you specify, and generates one reordered output file.

For example, the following list represents a file of records indicating temperature and humidity level at 2:00 p.m. for one week. Each record contains three fields: date, temperature, and humidity level.

Date	Temp	Humidity
9-15-80	75	55
9-16-80	68	50
9-17-80	70	90
9-18-80	78	60
9-19-80	83	50
9-20-80	80	70
9-21-80	85	80

You can create a new file that contains the same records in order from low to high temperature by specifying the temperature field as the key field in your sort. After sorting, the new file looks like this:

Date	Temp	Humidity
9-16-80	68	50
9-17-80	70	90
9-15-80	75	55
9-18-80	78	60
9-20-80	80	70
9-19-80	83	50
9-21-80	85	80

You run SORT at a terminal by using the SORT command line. Suppose you have a file of sales records like the ones below, with the name SALES.DAT. Each record contains six fields: date of sale, department code, salesperson, account number, customer name, and amount of sale. The number ranges below the list of records indicate the position and size of each information field within the record.

Date	Dpt	Salesp	Acct	Cust-Name	Amt
091580	25	Fielding	980342	Coolidge Carol	24999
091580	25	Sanchez	643881	McKee Michael	2499
091580	25	Bradley	753735	Rice Anne	10875
091580	19	Arndt	166392	Wilson Brent	1298
091580	28	Meredith	272731	Karsten Jane	4000
091580	25	Bradley	829582	Olsen Allen	3350
091580	19	ErKKila	980342	Coolidge Carol	7200
1–7	8–10	11–21	22–28	29–58	59–65

You want to rearrange the sales records so that they are in alphabetical order by customer name. The information field containing customer name is the key field, or "key," for your sort. As part of the SORT command, you must specify (1) the initial position of the key field in each record and (2) the length of the key field. You must also provide the names of (1) the file you want to sort and (2) the sorted file you create. You name the new file BILLING.LIS. The command you type at the terminal is:

```
$ SORT/KEY=(POSITION=29,SIZE=30) SALES.DAT BILLING.LIS ↵
```

SORT also accepts abbreviated command lines such as:

```
$ SORT /K=(PO=29,SI=30) SALES.DAT BILLING.LIS RET
```

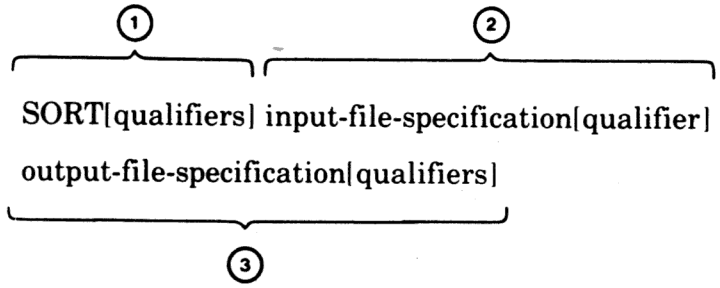
You can now display or print the new file of sorted records by using the TYPE or PRINT command.

```
$ TYPE BILLING.LIS RET

091580 19 Erkkila     980342 Coolidge Carol          7200
091580 25 Fielding    980342 Coolidge Carol         24999
091580 28 Meredith    272731 Karsten Jane            4000
091580 25 Sanchez     643881 McKee Michael           2499
091580 25 Bradley     829582 Olsen Allen             3350
091580 25 Bradley     753735 Rice Anne              10875
091580 19 Arndt       166392 Wilson Brent            1298
```

1.1.1 The SORT Command

The SORT command has three parts, separated by spaces.



1. Command name (SORT)

   SORT is the command name that invokes the VAX–11 SORT utility. Command name qualifiers describe the key(s) and specify sort options such as sorting process and number of work files. Use a slash (/) before any qualifiers that follow SORT. Of the eight qualifiers SORT accepts, /KEY is the only mandatory one.

   The SORT command needs two parameters:

   - The name of the file to be sorted (the input file)
   - The name you give the new, reordered file (the output file)

2. Input file parameter

   This parameter specifies the file or files you want to sort. You can sort up to 10 input files into one output file.

3. Output file parameter

   This parameter specifies the sorted file you want to create. Its qualifiers can request characteristics for the sorted output file. Specify only one output file.

1.1.2 SORT Qualifiers

Eight SORT qualifiers are available. You must always specify the /KEY qualifier. /KEY accepts several subqualifiers following an equals sign. You list these inside parentheses and separate them by commas. The SORT command for the sales records, for example, uses POSITION and SIZE subqualifiers:

```
$ SORT/KEY=(POSITION=29,SIZE=30) SALES.DAT BILLING.LIS RET
```

POSITION and SIZE are the most important key subqualifiers.

POSITION = n

Here n stands for the position within the record of the first byte in the key field. (The first position in the record is considered 1.) This position was established when the file was created; you can often determine it by examining a record from the file.

Always specify the POSITION subqualifier.

SIZE = n

Here n stands for the length of the key field in characters, bytes, or digits. (The unit depends on whether the data type of the key field is character, binary, or decimal.) Omit the SIZE subqualifier only for floating point data types, which have fixed sizes.

Any portion of a record can be a key. If you specify more than one key field, additional keys create suborders for records having the same primary key. You can sort records by as many as ten key fields with one command.

Suppose you want to rearrange the sales records that were sorted in Section 1.1 to show sales by department and by each salesperson in a department. Department code is the primary key and salesperson the secondary key. The command is:

```
$ SORT/KEY=(POSITION=8,SIZE=3)/KEY=(POSITION=11,SIZE=11) SALES.DAT
DEPT.LIS RET
```

Notice that you specify /KEY= followed by the necessary subqualifiers for each key by which you sort. Here is the reordered file you create with this command line:

```
091580 19 Arndt       166392 Wilson Brent          1298
091580 19 Erkkila     980342 Coolidge Carol        7200
091580 25 Bradley     829582 Olsen Allen           3350
091580 25 Bradley     753735 Rice Anne            10875
091580 25 Fielding    980342 Coolidge Carol       24999
091580 25 Sanchez     643881 McKee Michael         2499
091580 28 Meredith    272731 Karsten Jane          4000
```

Records in the new file are in order by department code, the primary key. Notice that when more than one record has the same department code, the sort creates a suborder based on the secondary key, salesperson. If the command had specified a third key, another suborder could have been created for Bradley's two sales records.

/KEY needs additional subqualifiers if:

- Key field data type is not character
- You list keys in an order different from their priority in the sort
- You want the sorted file in descending order (that is, highest to lowest number or Z to A)

Section 2.1.2 describes the subqualifiers you use in these cases.

1.1.3 Input File Parameter

SORT accepts up to ten input files. Separate multiple input file specifications by commas. For example, if you want to sort two days' files of sales records by customer name, type:

```
$ SORT/KEY=(POSITION=29,SIZE=30) SALES1.DAT,SALES2.DAT BILLING.LIS RET
```

The input file parameter accepts one qualifier, which you must use if your file is not on disk or ANSI magnetic tape. Section 2.1.1 describes this qualifier.

1.1.4 Output File Parameter

The command string may have only one output file specification. You need qualifiers for this parameter only if:

- You want your output file to have a different format, organization, or size from your input file(s)
- You are overwriting a file
- You want contiguous blocks for your output file

Section 2.1.3 explains the qualifiers you use in these circumstances.

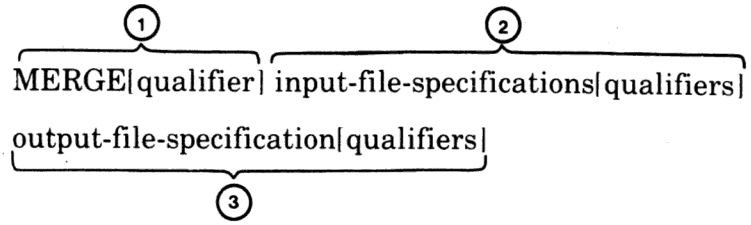
1.2 Merging Records

The MERGE utility takes two to ten similarly sorted input files, merges them according to the key field(s) you specify, and generates a single output file. Note that input files to MERGE must be in sorted order.

A central office might receive files of sorted sales records from eight stores, for example, and merge these files to create a single file. Or one store might merge sorted sales records each day with its main file.

1.2.1 The MERGE Command

The MERGE command has three main parts, separated by spaces.



1. Command name (MERGE)

   MERGE is the command name that invokes the VAX–11 MERGE utility. It takes three qualifiers, one of which describes the key(s) by which you want to merge your records.

   The MERGE command needs two parameters:

   - A list of file specifications of files to be merged
   - A name for the one merged output file

2. Input file parameter

   This parameter specifies the sorted files you want to merge. You must specify at least two and not more than ten files, and key(s) must be the same in all files. Separate input file specifications with commas.

3. Output file parameter

   This parameter specifies the merged file you want to create. Its qualifiers can request characteristics for the merged output file. The command string may have only one output file specification.

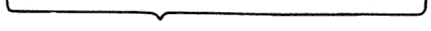
The following example of a merge operation continues the sales records example of Section 1.1. The central store has received the file of sorted sales records (now renamed STORE1.FIL) and wants to merge them with a similarly sorted file from another store (STORE2.FIL). The two files, both in order by customer name, are shown below.

STORE1.FIL

```
091580 19 ErkKila     980342 Coolidge Carol          7200
091580 25 Fielding    980342 Coolidge Carol         24999
091580 28 Meredith    272731 Karsten Jane            4000
091580 25 Sanchez     643881 McKee Michael           2499
091580 25 Bradley     829582 Olsen Allen             3350
091580 25 Bradley     753735 Rice Anne              10875
091580 19 Arndt       166392 Wilson Brent            1298
```

STORE2.FIL

```
091580 20 OConnor     358419 Beaulieu Ronald         1598
091580 04 Docus       980342 Coolidge Carol        575500
091580 25 Fielding    669011 Fernandez Felicia      12000
091580 35 Leith       848105 Kingsfield Stanley      5550
091580 04 Kramer      561903 Landsman Melissa      230000
091580 20 OConnor     643881 McKee Michael            995
091580 19 ErkKila     454389 VanDerling Julie        5480
```



The command to merge the two files is:

```
$ MERGE/KEY=(POSITION=29,SIZE=30) STORE1.FIL,STORE2.FIL CENTR.FIL ↵
```

Use TYPE or PRINT to see the merged file.

```
$ TYPE CENTR.FIL ↵

091580 20 OConnor     358419 Beaulieu Ronald                1598
091580 19 ErkKila     980342 Coolidge Carol                 7200
091580 25 Fielding    980342 Coolidge Carol                24999
091580 04 Docus       980342 Coolidge Carol               575500
091580 25 Fielding    669011 Fernandez Felicia             12000
091580 28 Meredith    272731 Karsten Jane                   4000
091580 35 Leith       848105 Kingsfield Stanley             5550
091580 04 Kramer      561903 Landsman Melissa             230000
091580 25 Sanchez     643881 McKee Michael                  2499
091580 20 OConnor     643881 McKee Michael                   995
091580 25 Bradley     829582 Olsen Allen                    3350
091580 25 Bradley     753735 Rice Anne                     10875
091580 19 ErkKila     454389 VanDerling Julie               5480
091580 19 Arndt       166392 Wilson Brent                   1298
```

1.2.2 MERGE Qualifiers

Three MERGE command qualifiers are available. You must always specify the /KEY qualifier.

/KEY subqualifiers are the same as for SORT keys: you must specify POSITION and, except for key data types with fixed sizes, SIZE. POSITION is the position within the record of the first byte in the key field (where the first position in the record is considered 1). SIZE is the length of the key field in characters, bytes, or digits.

/KEY subqualifiers follow the equals sign and are placed inside parentheses separated by commas. The simple merge just shown uses these two key subqualifiers:

```
$ MERGE/KEY=(POSITION=29,SIZE=30) STORE1.FIL,STORE2.FIL CENTR.FIL ↵
```

Specify additional /KEY subqualifiers if:

- Key field data type is not character
- You list keys in an order different from their priority in the merge
- Input file records have been sorted in descending order

Section 2.2.2 explains the subqualifiers you use in these cases.

1.2.3 Input File Parameter

The input file specification parameter accepts one qualifier. You need this qualifier if your file is not on disk or ANSI magnetic tape.

If used, this qualifier must apply to all input files.

See Section 2.2.1 for more details.

1.2.4 Output File Parameter

Just as for SORT, you need qualifiers for this MERGE parameter only if:

- Your output file has a different format, organization, or size from your input files
- You are overwriting a file
- You want contiguous blocks for your output file

Section 2.2.3 explains the qualifiers you use in these circumstances.

1.3 Running Batch SORT and MERGE

You can run your sort or merge task as a batch job. Batching frees your terminal for other work after you submit the job to the system for processing. Consider running frequent or lengthy sort and merge tasks as batch jobs.

To prepare a batch job, you first create a command procedure. To run the sort of sales records shown in Section 1.1 as a batch job, create your command procedure using a text editor or, as in this example, the CREATE command:

```
$ CREATE SALES.COM RET
```

Type in the following commands to sort the records in SALES.DAT by customer name and print out the reordered file:

```
$ SORT/KEY=(POSITION=29,SIZE=30) SALES.DAT BILLING.LIS
$ PRINT BILLING.LIS
^Z
```

Once you have created the command procedure, you execute it by using the SUBMIT command:

```
$ SUBMIT SALES.COM RET
```

The system assigns the batch job an identification number and enters it in the batch job queue. Immediately after you type the SUBMIT command, the system displays a message giving you the job identification number. For example:

```
Job 212 entered on queue SYS$BATCH
```

The system also creates a log file (in this example, SALES.LOG) to which it writes all output from the command procedure. When the batch job completes, this file is printed out and then deleted. The printed output from this batch job, then, is the sorted file, BILLING.LIS, and the batch job log file, SALES.LOG.

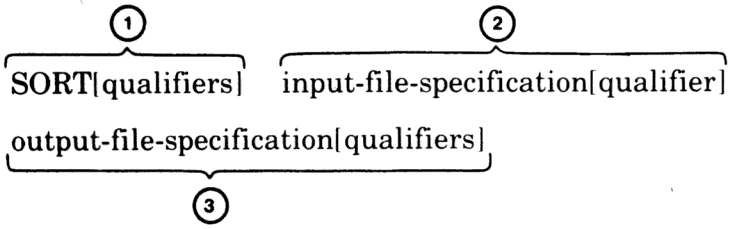
You can submit multiple command procedures and qualify the SUBMIT command in various ways. For additional information, see the *VAX/VMS Command Language User's Guide.*

Chapter 2
Qualifying SORT and MERGE Commands

This chapter describes the command qualifiers, subqualifiers, and parameters you specify when you use the SORT or MERGE command. You can specify these instructions interactively at a terminal. You can also enter them into a batch command procedure (see Chapter 1) or, for SORT only, a specification file (see Section 2.4). If you want only a brief description of SORT and MERGE qualifiers, see Section 2.3.

2.1 Qualifying the SORT Command

The SORT command has three parts, separated by spaces:



Sections 2.1.1. to 2.1.3. describe these three parts. The section describing the second part of the command line, input file specification, comes first because you need to consider the input file before you can determine what to specify for SORT command qualifiers and output file specification qualifiers.

For quick reference, Figure 2–1 summarizes the SORT command qualifiers, subqualifiers, and parameters.

2.1.1 Input File Parameter

Before entering the command line, first prepare the information SORT needs. Begin by determining the specification(s) of the file(s) you plan to sort. You provide this information in the second part of the command string as the input file parameter. Usually file name and type are an adequate file specification. (See the *VAX/VMS Command Language User's Guide* if you need more information on file specification.)

If you do not provide an input file specification, DCL prompts with:

```
$_File:
```

Default file type is DAT.

You can sort up to ten files at once; separate their file specifications by commas. Multiple input files must have the same file organization, record format, and key description. VAX–11 SORT accepts all VAX–11 RMS files. That is, it accepts sequential, relative or indexed-sequential data files on one or more volumes, containing records of fixed, variable, or variable with fixed-length control format. Multiple input files with VFC record format must all have the same length control portion. SORT does not support FORTRAN segmented files.

Suppose you plan to sort a file of records named READ.FIL, which gives results from reading tests taken by fourth-, fifth-, and sixth-grade students (see Figure 2–2). Specify READ.FIL as the input file parameter.

```
$ SORT[QUALIFIERS] READ.FIL OUTPUT.FIL
```

If you want to sort two files of similar records (for example, different schools in one system), list the files in this way:

```
$ SORT[QUALIFIERS] READ1.FIL,READ2.FIL OUTPUT.FIL
```

Before carrying out a sort operation, VAX–11 SORT needs to know the size of the file(s) it is about to sort and the size of the longest record in the file(s). For files on disk or ANSI magnetic tape, VAX–11 Record Management Services (RMS) automatically provides information about record and file size to the SORT utility program. But if your input file is on an input device from which RMS cannot get file and record size, such as a terminal, you must provide this information by using the following qualifier and subqualifers immediately following input file specifications:

/FORMAT = (RECORD_SIZE = n,FILE_SIZE = n)

RECORD_SIZE = n
: You specify record size by giving the longest record length (LRL) in bytes. The longest record length allowed is 32,767 bytes (16,383 for relative files). These totals include control bytes for VFC files. For multiple files, LRL is the longest record length of all records in all files.

  For additional information on determining the LRL, refer to the $FAB MRS parameter in the *VAX–11 Record Management Services Reference Manual.*

  In the case of an RMS error, you can use this subqualifier to override the record size normally retrieved from RMS.

FILE_SIZE = n
: You specify file size in blocks. Maximum file size accepted is 4,294,967,295 blocks. For multiple input files, n is the total size of all files. If file size is not provided by RMS or the user, SORT assumes 1000 blocks. SORT uses file size information to determine the size of its work files.

Figure 2–1: SORT Command Summary

Command Format: $SORT/QUALIFIERS INPUT.FIL/QUALIFIERS OUTPUT.FIL/QUALIFIERS

$ SORT

SORT Qualifiers	Subqualifiers and Values	Notes
/KEY=	([**CHARACTER** / BINARY / F_FLOATING / D_FLOATING / G_FLOATING / H_FLOATING / ZONED / DECIMAL / PACKED_DECIMAL]	/KEY is not required with /SPECIFICATION
	[,**SIGNED** / ,UNSIGNED]	Use with binary data
	[,**TRAILING_SIGN** / ,LEADING_SIGN]	Use with decimal data
	[,**OVERPUNCHED_SIGN** / ,SEPARATE_SIGN]	Use with decimal data
	,POSITION={1-32767}	
	[,SIZE={1-255 for CHARACTER data / 1,2,4,or 8 for BINARY data / 1-31 for DECIMAL data}]	Required for all data types except floating point
	[,NUMBER={1-10}]	
	[,**ASCENDING** / ,DESCENDING])	
[/PROCESS=	{**RECORD** / TAG / ADDRESS / INDEX}]	
[/STABLE]		
[/COLLATING_SEQUENCE=	{**ASCII** / EBCDIC}]	
[/WORK_FILES=	{0,**2**-10}]	
[/STATISTICS]		
[/SPECIFICATION	[=file-specification]]	
[/RSX11]		Requires PDP-11 SORT command switches

(continued on next page)

Notation used:
- Optional qualifiers are enclosed in brackets [ ]. Use one or none from a list enclosed in brackets.
- For values enclosed in braces { } you must select one.
- Defaults are shown in bold type.

Figure 2–1: SORT Command Summary (Cont.)

INPUT.FIL/QUALIFIERS		
Input-file-specification(s)		
Input File Qualifiers	**Subqualifiers and Values**	**Notes**
[/FORMAT=	(RECORD_SIZE={1-32767} ,FILE_SIZE={1-4294967295})]	Required for files not on disk or ANSI magnetic tape Use with above files Default is 1000 blocks
OUTPUT.FIL/QUALIFIERS		
Output-file-specification		
Output File Qualifiers	**Subqualifiers and Values**	**Notes**
[/FORMAT=	[FIXED=[1-32767] VARIABLE=[1-32767] CONTROLLED=[1-32767]]]	
	[,SIZE={1-255}]	Use with CONTROLLED records Default is 2 bytes
	[,BLOCK_SIZE={18-65535}]	Use with tape files
[/SEQUENTIAL /RELATIVE /INDEXED_SEQUENTIAL]		
[/ALLOCATION=	{1-4294967295}]	Required with /CONTIGUOUS
[/CONTIGUOUS]		Use with /ALLOCATION
[/LOAD_FILL]		Use with INDEXED files
[/OVERLAY]		Required for INDEXED files
[/BUCKET_SIZE=	{1-32}]	Use with disk files

If you specify only one /FORMAT subqualifier, you can omit the parentheses.

If your two files of reading test records are not on disk or ANSI magnetic tape, your input file parameter looks like this example:

```
READ1.FIL,READ2.FIL/FORMAT=(RECORD_SIZE=75,FILE_SIZE=6)
```

Notice that you specify the /FORMAT qualifier once, following file specifications for all input files.

2.1.2 SORT Command Qualifiers

As the first part of the command string, specify the command SORT followed by the qualifier(s) appropriate to your sorting task. SORT accepts eight qualifiers:

/KEY
/PROCESS
/STABLE
/COLLATING_SEQUENCE
/WORK_FILES
/STATISTICS
/SPECIFICATION
/RSX11

The rest of this section explains these qualifiers. List the qualifiers in any order following the command SORT.

/KEY

After you have determined the needed input-file information, decide which data field(s) in your records you want to arrange in sequence. You can sort by one key field (student name, for example), by two (grade and name), or by any number of keys up to ten. If you select multiple keys, decide which is primary, which secondary, and so on.

Suppose you want to arrange the reading scores from highest to lowest score for each class within each grade. Primary key is grade; secondary key is class (indicated by teacher name); tertiary key is score.

For each key, you use key subqualifiers to provide the sort utility with information about:

- The way key field data is stored in your records

  (DATA TYPE, POSITION, SIZE)

- Your decisions for the sort

  NUMBER, ASCENDING/DESCENDING)

Figure 2–2: A Sample Sort

1. Observe the input file of reading test records (READ.FIL):

Date	Gr	Teacher	Student	Score	Level	Test
03-10-80	4	Anderson	Brock Diane	470	+ 2	Bates-McK
03-10-80	6	ONeill	Schillaci Rosemary	620	- 3	Bates-McK
03-10-80	5	Walczak	Vaillancourt Roger	560	+ 1	Bates-McK
03-10-80	6	MacLeod	Ecklund Kristine	740	+ 9	Bates-McK
03-10-80	5	Reiner	Sanchez Juan	590	+ 4	Bates-McK
03-10-80	4	Marquez	Bellermann Robert	470	+ 2	Bates-McK
03-10-80	6	ONeill	Hartman Harold	590	- 6	Bates-McK
03-10-80	5	Reiner	Wu Karen	610	+ 6	Bates-McK
03-10-80	4	Anderson	Braddon Michael	460	+ 1	Bates-McK
03-10-80	6	MacLeod	Parker Ruthann	650	0	Bates-McK
03-10-80	4	Marquez	Olanbak Raymond	650	+20	Bates-McK
03-10-80	5	Walczak	Cochrane Nancy	510	- 4	Bates-McK
03-10-80	4	Marquez	Hanley Vivian	350	-10	Bates-McK
03-10-80	6	ONeill	Cote Yvonne	800	+15	Bates-McK
03-10-80	4	Anderson	Nolan Keith	530	+ 8	Bates-McK
03-10-80	5	Reiner	Rodino Madeleine	540	- 1	Bates-McK
03-10-80	6	ONeill	Grodinski Richard	650	0	Bates-McK
03-10-80	5	Reiner	Kessler Peter	520	- 3	Bates-McK
03-10-80	5	Walczak	Levine Carol	440	-11	Bates-McK
03-10-80	4	Anderson	Tillotson Judith	440	- 1	Bates-McK
03-10-80	6	MacLeod	Delano Paul	690	+ 4	Bates-McK
03-10-80	4	Marquez	James Joseph	630	+18	Bates-McK
03-10-80	6	MacLeod	Fisher Carolyn	700	+ 5	Bates-McK
03-10-80	5	Walczak	Yuan Kim	560	+ 1	Bates-McK
03-10-80	5	Reiner	Lopez Edward	650	+10	Bates-McK
	11		14-28	54-46	59-61	

2. Enter this command to arrange records from highest to lowest score for each class within each grade:

```
$ SORT/KEY=(POSITION=11,SIZE=1)/KEY=(POSITION=14,SIZE=15)/KEY=(POSITION=54,
    SIZE=3,DESCENDING)   READ.FIL READ.LIS (RET)
```

3. The sorted output file (READ.LIS) is:

Date	Gr	Teacher	Student	Score	Level	Test
03-10-80	4	Anderson	Nolan Keith	530	+ 8	Bates-McK
03-10-80	4	Anderson	Brock Diane	470	+ 2	Bates-McK
03-10-80	4	Anderson	Braddon Michael	460	+ 1	Bates-McK
03-10-80	4	Anderson	Tillotson Judith	440	- 1	Bates-McK
03-10-80	4	Marquez	Olanbak Raymond	650	+20	Bates-McK
03-10-80	4	Marquez	James Joseph	630	+18	Bates-McK
03-10-80	4	Marquez	Bellermann Robert	470	+ 2	Bates-McK
03-10-80	4	Marquez	Hanley Vivian	350	-10	Bates-McK
03-10-80	5	Reiner	Lopez Edward	650	+10	Bates-McK
03-10-80	5	Reiner	Wu Karen	610	+ 6	Bates-McK
03-10-80	5	Reiner	Sanchez Juan	590	+ 4	Bates-McK
03-10-80	5	Reiner	Rodino Madeleine	540	- 1	Bates-McK
03-10-80	5	Reiner	Kessler Peter	520	- 3	Bates-McK
03-10-80	5	Walczak	Yuan Kim	560	+ 1	Bates-McK
03-10-80	5	Walczak	Vaillancourt Roger	560	+ 1	Bates-McK
03-10-80	5	Walczak	Cochrane Nancy	510	- 4	Bates-McK
03-10-80	5	Walczak	Levine Carol	440	-11	Bates-McK
03-10-80	6	MacLeod	Ecklund Kristine	740	+ 9	Bates-McK
03-10-80	6	MacLeod	Fisher Carolyn	700	+ 5	Bates-McK
03-10-80	6	MacLeod	Delano Paul	690	+ 4	Bates-McK
03-10-80	6	MacLeod	Parker Ruthann	650	0	Bates-McK
03-10-80	6	ONeill	Cote Yvonne	800	+15	Bates-McK
03-10-80	6	ONeill	Grodinski Richard	650	0	Bates-McK
03-10-80	6	ONeill	Schillaci Rosemary	620	- 3	Bates-McK
03-10-80	6	ONeill	Hartman Harold	590	- 6	Bates-McK

Figure 2–3: KEY Specifications

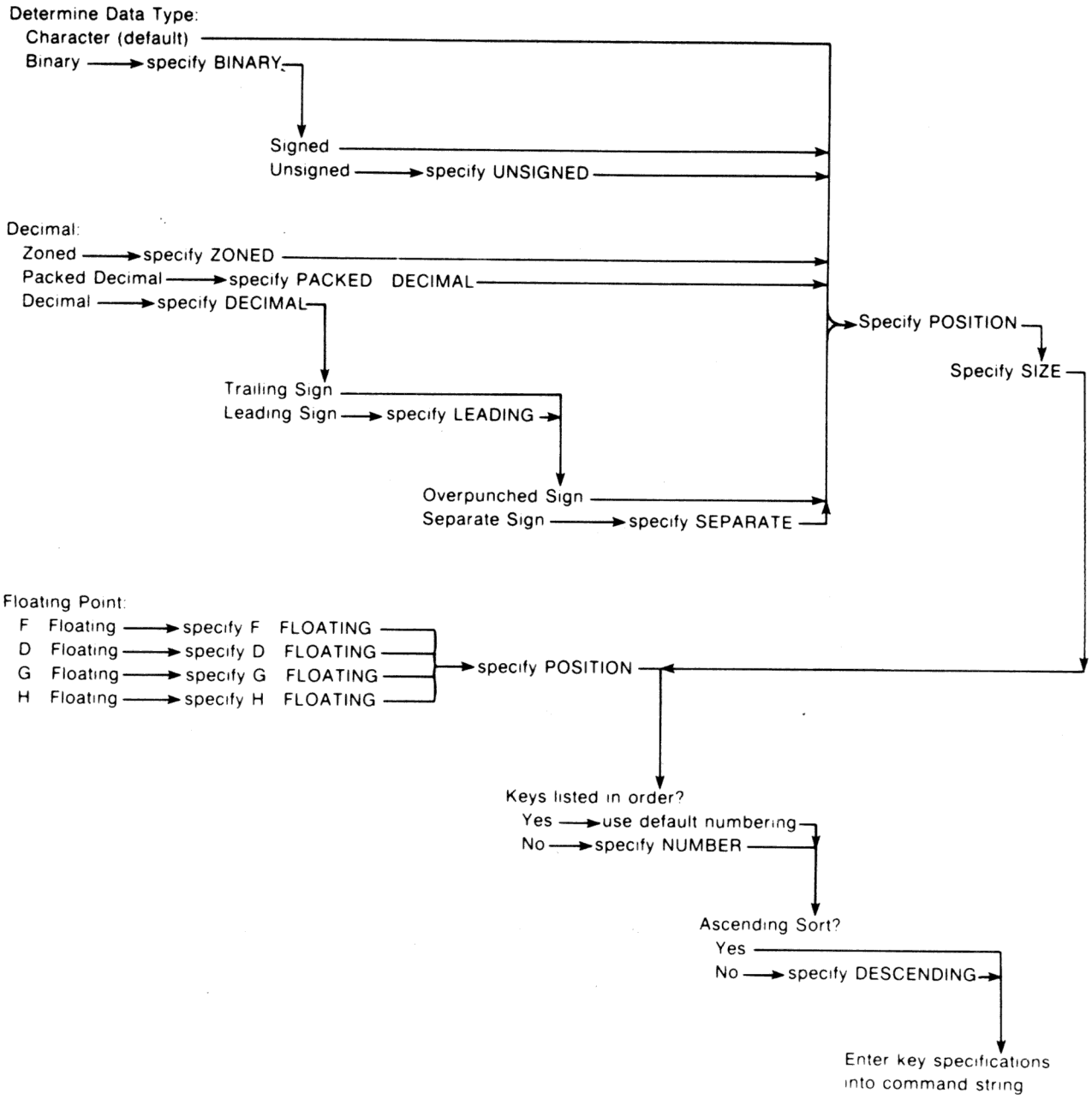


Figure 2–3 provides a chart for quick reference when you are specifying keys.

DATA TYPE

To provide SORT with the necessary information about your keys, you must first know the data type used for storing information in your key field. The program used to create the file(s) established the data type of each field in the record.

The data types recognized by VAX–11 SORT are:

CHARACTER (ASCII order or EBCDIC order)
BINARY (SIGNED or UNSIGNED)
F_FLOATING
D_FLOATING
G_FLOATING
H_FLOATING
ZONED
DECIMAL (LEADING or TRAILING, SEPARATE or OVERPUNCHED)
PACKED_DECIMAL

CHARACTER is the default data type. If the data in your key field is not stored as character, you must specify the data type.

In addition, if your key data type is unsigned binary, specify UNSIGNED. This subqualifier enables SORT to accept larger binary numbers; however, it treats all numbers as positive. SIGNED is the default.

Finally, if your key data type is decimal, you must determine two more facts:

- Whether the sign of the decimal data key is stored at the beginning or end of the key
- Whether the sign is overpunched or is separated from the decimal

TRAILING_SIGN or LEADING_SIGN:
If the sign is stored at the end of the key, you do not need to type in any keyword. TRAILING_SIGN is the default.

If the sign is stored at the beginning of the key, you must specify LEADING_SIGN.

OVERPUNCHED_SIGN or SEPARATE_SIGN:

If the sign of your decimal key is overpunched, you do not need to specify anything. OVERPUNCHED_SIGN is the default.

If the sign is separated from the decimal, you must specify SEPARATE_SIGN.

In the records in Figure 2–2, grade, teacher, and score are all stored as character data. For this sort, then, you do not need to use any data type subqualifiers.

POSITION

For every key, you must use POSITION = n. Here n is the position of the first byte of the key field within the record. The position of the first byte in a record is considered position 1. Positions of data fields were established when the file was created.

In Figure 2–2, grade begins in position 11, teacher position 14, and score position 54.

SIZE

Unless your data type is one of the floating point data types (for which size is fixed), you must specify SIZE = n, where n is the length of the sort key.

Depending on data type, you specify size in characters, bytes, or digits. If sort key data type is character, key size in characters must not be more than 255. If data type is binary, key size in bytes must be 1, 2, 4, or 8 (that is, byte, word, longword, or quadword). If the data type is any of the decimal types, key size in digits must not exceed 31. The total of all key field sizes must not be more than 255 bytes.

Note that for decimal data, length is measured in digits, not in bytes. Thus, if the sign is stored in a separate byte (as it is for Leading Separate and Trailing Separate decimal strings), that byte is not counted in figuring size. However, it is counted in figuring position.

Remember this method of using digits to determine decimal size when key data type is packed decimal, which stores two decimal digits in one byte.

The data field containing level in the reading-score records might contain the information stored as decimal with a leading, separate sign. In this case, key specifications are:

```
/KEY=(POSITION=59,SIZE=2,DECIMAL,LEADING,SEPARATE)
```

When sorting data not stored as character, remember that certain non-character data should not be printed without being converted by a program.

In the example above, size for grade key is 1, for teacher 15, and for score 3. At this point, after you have considered data type, position, and size, key specifications are:

```
SORT/KEY=(POS=11,SIZE=1)/KEY=(POS=14,SIZE=15)/KEY=(POS=54,SIZE=3)
```

NUMBER

If you are sorting on several keys, and you list them in an order different from the order of their priority in the sort, then you must specify the place of each key in the sort by using NUMBER=n. Acceptable values for n are 1–10, where 1 is the primary sort key, 2 is the secondary sort key, and so on. If you do not specify NUMBER, SORT assigns NUMBER=1 for the first key you list, and the NUMBER value of the previous key plus 1 for each following key.

Of course, if you define your keys in order of decreasing priority, you can omit this subqualifier; SORT assigns the appropriate numbers.

ASCENDING/DESCENDING

SORT normally arranges records in ascending order (that is, A to Z or lowest to highest number). ASCENDING is the default. If you want to sort any key in descending order, you must indicate this by specifying DESCENDING as a key subqualifier.

In the example, you want to arrange reading scores so the highest score in each class is first. This is a descending sort, and you must use the DESCENDING subqualifier with this key.

You must specify these key subqualifiers and values for each key field by which you sort your file. List key subqualifiers inside parentheses following /KEY= and separate them with commas. You can list them in any order.

/PROCESS

SORT offers four processes by which you can sort your records:

/PROCESS=RECORD
/PROCESS=TAG
/PROCESS=ADDRESS
/PROCESS=INDEX

Depending on which process you select, your output file contains the records themselves (record or tag sort) or an index to the records (address or index sort). You must submit this index to a program for further processing.

The four processes differ also in input and output device requirement, number of input files accepted, and processing method. You must understand these differences to select the process best suited to your sorting task. Table 2–1 shows these differences.

Table 2–1: SORT Processes

Process	Input Device	Input File	Processing Method	Output File Content	Output Device
Record	any VAX/VMS input device	single or multiple files	keeps record intact throughout sort	complete records	any VAX/VMS output device
Tag	disk only	one file	sorts only key(s), then reaccesses input file records to create output file	complete records	any VAX/VMS output device
Address	disk only	one file	sorts only key(s)	binary RFAs only (Record File Address)	any device that accepts binary data
Index	disk only	one file	sorts only key(s)	keys and binary RFAs	any device that accepts binary data

Selecting a Sort Process

1. Consider first how you will use the output file.

   - Record and tag sort generate files containing the entire records. These reordered files are ready to be printed out and distributed, or stored.
   - Address sort creates a list of pointers to the records in the input file. This list takes the form of binary Record File Addresses (RFAs). A program can use the list of pointers to access records in the rearranged order for further processing or later output.
   - Index sort creates an output file containing both RFAs and key fields. If your program needs key field content for decision during future processing, select index sort rather than address sort. Because the output file from index sort contains key field data, the processor does not have to get it from auxiliary storage. (Do not confuse this index file generated by SORT with RMS indexed file organization.)

     You may need to arrange the same data in several ways for different purposes. Sales records, for example, can provide the basis for billing customers, updating inventory, and keeping track of sales by department and salesperson. Instead of storing several versions of the same file, you can store several output files from address or index sort in much less space. Then use these files to access the records in the main file in the order you want.

2. Also consider the temporary storage space you have available for the sort.

   - Tag sort uses less temporary storage space than record sort, which, because it keeps the record intact during the sort, can take up much work space for large files.
   - Address and index sort use little space.

3. Input and output device and number of input files can also affect your selection of sort process.

   - Record sort is the only process that can accept input from cards and magnetic tape as well as disk.
   - Output from address and index sort must go to a device that accepts binary data.
   - Only record sort accepts multiple input files.

4. Finally, consider differences in speed.

   - If you plan to retrieve the sorted records in order at some point in your operation, record sort will usually be the fastest process.
   - Because tag sort moves only keys instead of complete records, it can be faster than record sort when record size is very large and key size is small. Tag sort can also be faster for exceedingly large files and devices with a fast seek time. In most cases, however, the time tag sort takes to reaccess the input file makes it slower than record sort.
   - Address and index sort are the fastest processes (but of course do not output the records).

Record sort is the default sort process.

For the reading score sort (Figure 2–2), you want to print out the reordered file, which means you should select either record or tag sort. You have adequate temporary storage space, and record size is not large enough to suggest that tag sort might be faster than record sort. The best choice then is record sort, the default process.

If instead of printing out the sorted records, you want to submit them to a program for determining median score in each class and grade, select an index sort and include in your command the qualifier: /PROCESS= INDEX. Note, however, that if you have multiple input files, you must select record sort.

/STABLE

If you request stable sort, SORT keeps records with equal keys in their original order. Otherwise, their order in the output file is unpredictable. Because this capability takes additional memory, use it only when your sorting task requires keeping records with the same keys in the same order.

If you use /STABLE, maximum key size is 251 instead of 255.

/COLLATING_SEQUENCE

SORT arranges characters in ASCII sequence. If you want SORT to arrange characters in EBCDIC sequence (for example, a program may require an input file in EBCDIC sequence), specify /COLLATING_SEQUENCE=EBCDIC. Note that the characters remain ASCII representation; only the order changes.

COLLATING_SEQUENCE=ASCII is the default.

/WORK_FILES

You can instruct SORT to use no work files or any number from 2 through 10. SORT does not create work files until it needs them. Then, unless instructed differently, it creates two temporary work files for your sort, determining their size from the size of your input file(s).

Usually there is no advantage to using more than two work files. However, if you have plentiful work space on disk and want a faster sort, you can change the location of the work files. See Chapter 4 for more information.

You assign a work file to a device other than your current default device by typing:

```
$ ASSIGN (device): SORTWORKn
```

SORTWORKn is a logical name for the work file, where n indicates the number of the work file. Acceptable values are 0 through 9.

/STATISTICS

If you want a statistical summary at the end of your sort operation, use this qualifier. The statistical summary includes, among other things, the number of records read, sorted, and output, and the maximum working set used. Section 4.2 explains how the information from this summary can help you improve your SORT efficiency.

/SPECIFICATION

A specification file is a file containing the sort instructions usually conveyed by the SORT qualifiers. If you use this method to perform your sorting task, in the SORT command you need only the qualifier /SPECIFICATION[=file-specification] in the SORT command. See Section 2.4 for an explanation of specification file.

/RSX11

You can invoke SORT–11 instead of VAX–11 SORT by specifying this qualifier. If you select this option, use DCL (DIGITAL Command Language) syntax with PDP-11 SORT qualifiers. Refer to the *PDP–11 SORT Reference Manual* for more information.

2.1.3 Output File Parameter

The third part of the SORT command string is the file specification of your output file. Specify only one output file. The output file in Figure 2–2 is READ.LIS.

If you do not provide an output file specification, DCL prompts with:

```
$_Output:
```

If you omit file type from your file specification, SORT defaults to the file type of the first input file.

The output file parameter accepts seven qualifiers (one of which has three subqualifiers). Many sort operations do not use any of these qualifiers. The following list gives conditions under which you need a qualifier and the appropriate qualifier or subqualifier. The rest of this section explains how to use each qualifier. List any qualifiers immediately after output file specification.

You need a qualifier if:

- You want output file record format to differ from input file record format:

  /FORMAT = (FIXED = n
  (VARIABLE = n
  (CONTROLLED = n

- You want the fixed portion of a controlled record to differ from input file fixed portion size:

  SIZE = n

- You want the block size of a magnetic tape output file to differ from input file block size:

  BLOCK_SIZE = n)

- You want the file organization of your output file to differ from that of the input file:

  /SEQUENTIAL
  /RELATIVE
  /INDEXED_SEQUENTIAL

- You want to allocate a certain number of blocks on disk for your output file:

  /ALLOCATION = n

- You want contiguous allocation of blocks for the output file:

  /CONTIGUOUS

- You are sorting an indexed file and want SORT to load buckets according to the fill size established when the file was created:

  /LOAD_FILL

- You want to overwrite an existing empty file which has the same name as the output file:

  /OVERLAY

- You want to specify RMS bucket size for the output file:

  /BUCKET_SIZE = n

The following paragraphs explain each of these qualifiers further.

/FORMAT = (FIXED = n
(VARIABLE = n
(CONTROLLED = n

Use this qualifier if you want your output file record format to differ from input file record format. You can specify fixed-length records (FIXED), variable-length records (VARIABLE), or variable with fixed-length control records (CONTROLLED). If you omit this qualifier, output record format is the same as input file record format if you are performing record or tag sort, and FIXED for address or index sort.

You can also specify with each format subqualifier the longest record length (LRL) in bytes of the output records (n). LRL is optional. The longest record length allowed is 32,767 bytes (16,383 bytes for relative files). These totals include control bytes. For additional information on determining LRL, refer to the $FAB MRS parameter in the *VAX–11 Record Management Services Reference Manual.*

SIZE = n

Include this qualifier for CONTROLLED records only. It specifies the byte size of the fixed portion. Default size is that of the input file if it contains CONTROLLED records, or 2 bytes if input file record format is not CONTROLLED. Maximum size is 255 bytes.

BLOCK_SIZE = n)

Use this qualifier with magnetic tape files only. Specify the block length of the output file in bytes. Default value is the block size established at tape mounting time, or if no size was specified, the block size of the input file tape. Block size must be 18 to 65,535 bytes.

NOTE

To assure correct data interchange with other DIGITAL systems, specify a block size of not more than 512 bytes. To assure compatibility with most non–DIGITAL systems, the block size should not exceed 2048 bytes.

/SEQUENTIAL
/RELATIVE
/INDEXED_SEQUENTIAL

You can use one of these qualifiers to specify output file organization. If you omit this qualifier, output file organization is the same as input file organization for record or tag sort, and /SEQUENTIAL for address or index sort.

If you specify /INDEXED_SEQUENTIAL (or if the output file is INDEXED_SEQUENTIAL by default), the output file must exist, its organization must be indexed sequential, and it must be empty; therefore, you must also specify /OVERLAY.

/ALLOCATION = n

You can allocate a certain number of 512–byte blocks of disk space for your output file. Acceptable values are from 1 to 4,294,967,295.

If you plan to extend a file, or if you want a contiguous allocation of blocks, use this qualifier.

/CONTIGUOUS

You can require contiguous allocation of blocks for your output file by specifying /CONTIGUOUS. This qualifier must be used with /ALLOCATION. It is invalid if you have not specified /ALLOCATION, or if /ALLOCATION value is not large enough for total output and the file must be extended.

/LOAD_FILL

Use this qualifier only with INDEXED files. It loads the buckets according to the fill size established when the file was created, by using the $XABKEY DFL and $XABKEY IFL parameters. This procedure minimizes bucket splitting if many records are added later. See the *VAX–11 Record Management Services Reference Manual* for more information.

/OVERLAY

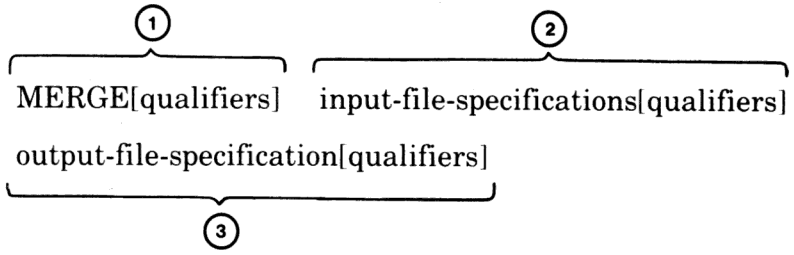
Use /OVERLAY to write the sorted file into an existing file. The existing file must be empty.

/BUCKET_SIZE = n

You can specify RMS bucket size (that is, the number of 512–byte blocks per bucket) for the output file by using this qualifier. If output file has the same file organization as input file, default value is the same as the input file bucket size. If output file organization is different, default value is 1. The maximum number of blocks per bucket is 32.

2.2 Qualifying the MERGE Command

The MERGE command enables you to combine up to ten similarly sorted files. Like the SORT command, the MERGE command has three parts, separated by spaces:



Sections 2.2.1 to 2.2.3 describe these three parts. The section describing the second part of the command line, input file specifications, comes first because you need to consider the input file before you can determine what to specify for MERGE command qualifiers and output file specification qualifiers. Figure 2–4 provides a summary of MERGE command qualifiers, subqualifiers, and parameters for quick reference.

2.2.1 Input File Parameter

Before entering the command line, prepare the information MERGE needs. Begin by determining the file specifications of the files you plan to merge. You provide this information in the second part of the command string as the input file parameter. You must give at least two file specifications and not more than ten. Separate file specifications by commas.

If you do not provide input file specifications, DCL prompts with:

```
$_File:
```

Usually you need only file name and type for each file. If any input file specification does not contain file type, MERGE defaults to DAT.

Files must have the same record and key characteristics. For example, you cannot merge the file READ.LIS (Figure 2–2) with a file of reading test records that have the key fields in different positions. Files of records, though, can contain varying data as long as key fields remain the same.

Input files must also have the same file organization. If they do not, you can convert them to the same organization by using the CONVERT/MERGE utility. See the *VAX–11 Record Management Services Reference Manual* on how to convert file organization.

The input file parameter accepts one qualifier. This qualifier accepts two subqualifiers. To this qualifier, place it after the list of input file specifications. This qualifier must apply to all input files.

Figure 2–4: MERGE Command Summary

Command Format: $MERGE/QUALIFIERS INPUT.FLS/QUALIFIERS OUTPUT.FIL/QUALIFIERS

$ MERGE

MERGE Qualifiers	Subqualifiers and Values	Notes
/KEY=	([**CHARACTER** | BINARY | F_FLOATING | D_FLOATING | G_FLOATING | H_FLOATING | ZONED | DECIMAL | PACKED_DECIMAL]	
	[,**SIGNED** | ,UNSIGNED]	Use with binary data
	[,**TRAILING_SIGN** | ,LEADING_SIGN]	Use with decimal data
	[,**OVERPUNCHED_SIGN** | ,SEPARATE_SIGN]	Use with decimal data
	,POSITION={1-32767}	
	[SIZE={1-255 for CHARACTER data | 1,2,4,or 8 for BINARY data | 1-31 for DECIMAL data}]	Required for all data types except floating point
	[,NUMBER={1-10}]	
	[,**ASCENDING** | ,DESCENDING])	
[/CHECK_SEQUENCE]		
[/COLLATING_SEQUENCE={**ASCII** | EBCDIC}]		

(continued on next page)

Notation used:
- Optional qualifiers are enclosed in brackets [ ]. Use one or none from a list enclosed in brackets.
- For values enclosed in braces { } you must select one.
- Defaults are shown in bold type.

Figure 2–4: MERGE Command Summary (Cont.)

INPUT.FLS/QUALIFIER		
Input-file-specifications		
Input File Qualifier	**Subqualifiers and Values**	**Notes**
[/FORMAT=	(RECORD__SIZE=|1-32767| [,FILE__SIZE=|1-4294967295|)]	Required for files not on disk or ANSI magnetic tape Use with above files Default is 1000 blocks
OUTPUT.FIL/QUALIFIERS		
Output-file-specification		
Output File Qualifiers	**Subqualifiers and Values**	**Notes**
[/FORMAT=	[FIXED=[1-32767] VARIABLE=[1-32767] CONTROLLED=[1-32767]] [,SIZE=|1-255|] [,BLOCK__SIZE=|18-65535|]]	Use with CONTROLLED records Default is 2 bytes Use with tape files
[/SEQUENTIAL /RELATIVE /INDEXED__SEQUENTIAL]		
[/ALLOCATION=	|1-4294967295|]	Required with /CONTIGUOUS
[/CONTIGUOUS]		Use with /ALLOCATION
[/LOAD__FILL]		Use with INDEXED files
[/OVERLAY]		Required for INDEXED files
[/BUCKET__SIZE=	|1-32|]	Use with disk files

/FORMAT = (RECORD_SIZE = n,FILE_SIZE = n)

The MERGE utility needs to know: (1) the size of the files it is about to merge and (2) the size of the longest record in those files. For files on disk or ANSI magnetic tape, RMS automatically provides information about record and file size to the MERGE utility program. But if your input file is on an input device from which RMS cannot get file and record size, such as a terminal, you must provide this information by using the following qualifier and subqualifiers immediately following input file specifications:

RECORD_SIZE = n — You specify record size by giving the longest record length (LRL) in bytes. The longest record length allowed is 32,767 bytes (16,383 bytes for relative files). These totals include control bytes. For multiple files LRL is the longest record in all input files.

For additional information on determining the LRL, refer to the $FAB MRS parameter in the *VAX-11 Record Management Services Reference Manual*.

In the case of an RMS error, you can use this subqualifier to override the record size normally retrieved from RMS.

FILE_SIZE = n — You specify file size in blocks. Here n is the total size of all input files. Maximum file size accepted is 4,294,967,295 blocks. If file size is not provided by RMS or the user, MERGE assumes 1000 blocks.

If you specify only one /FORMAT subqualifier, you can omit the parentheses.

2.2.2 MERGE Command Qualifiers

The MERGE command accepts three qualifiers:

/KEY
/CHECK_SEQUENCE
/COLLATING_SEQUENCE

You must always specify /KEY.

Keys must be the same in all files to be merged. MERGE key subqualifiers are the same as SORT key subqualifiers. Remember, you merge sorted files on keys already used for sorting. You should know how the files you plan to merge have been sorted. You determine data type, position, and size in the same way for both SORT and MERGE. The subqualifiers NUMBER and DESCENDING refer to the order in which keys have been sorted in your input files. See Section 2.1.2 for information about key subqualifiers.

Keys in MERGE input files must be arranged in the same order. For example, READ.LIS, the file sorted in Figure 2–2, arranged reading scores from highest to lowest score within class. You cannot merge READ.LIS with a file of similar records arranged from lowest to highest reading score if you plan to merge on that key. However, you can merge the two files by grade and teacher as long as these fields are arranged in similar order in both files.

/CHECK_SEQUENCE

If you use this qualifier, MERGE checks your input files to make sure they are in order. If a record is out of order, MERGE gives you the following message at the end of the merge operation:

```
%SOR-F-BAD_ORDER    Input file [n] is out of order
```

In place of n, MERGE specifies the number of the input file (1–10).

You must use /COLLATING_SEQUENCE = EBCDIC if sorted files are in EBCDIC sequence.

2.2.3 Output File Parameter

MERGE combines all input files into one merged output file. Give the output file specification as the third part of the MERGE command. Specify only one output file.

If you do not provide an output file specification, DCL prompts with:

```
$_Output:
```

If you omit file type from your file specification, MERGE defaults to the file type of your first input file.

The MERGE output file parameter accepts the same seven qualifiers as does the SORT output file parameter. Many merge operations do not need any of these qualifiers. You need one or more only if:

- You do not want to accept MERGE default values for organization, format, and size
- You are overwriting an existing file
- You want contiguous blocks for your output file

Use MERGE output file qualifiers in the same way as SORT output file qualifiers. See Section 2.1.3 for a description of these qualifiers.

2.3 Qualifier Summary

This section provides a brief description of the qualifiers you can use with the SORT or MERGE command. Default values are in bold type.

2.3.1 SORT/MERGE Command Qualifiers

/KEY = (subqualifier,subqualifier...)

The /KEY qualifier must be specified unless defined in a specification file. It defines a SORT or MERGE key, and may appear up to ten times in a command line.

Enclose the /KEY subqualifiers group in parentheses.

(POSITION = n

The POSITION subqualifier is mandatory. Here n specifies the position of the first byte of the key field within the record, where the first position in the record is considered 1.

SIZE = n

The SIZE subqualifier is mandatory except for the floating point data types. Here n specifies the length of the sort or merge key in characters, bytes, or digits, depending on the key field data type. If key data type is CHARACTER, key size in characters must not be more than 255. If data type is BINARY, key size in bytes must be 1, 2, 4, or 8 (byte, word, longword, or quadword). If data type is any of the DECIMAL types, key size in digits must not exceed 31. Note that for decimal data, SIZE is measured in digits, not bytes. The total of all key field sizes must not be more than 255 bytes.

CHARACTER
BINARY
F_FLOATING
D_FLOATING
G_FLOATING
H_FLOATING
ZONED
DECIMAL
PACKED_DECIMAL

This subqualifier indicates the data type used for storing information in the key field. CHARACTER is the default data type.

SIGNED
UNSIGNED

Use this subqualifier with binary data. It indicates whether a binary key is to be compared as a signed or an unsigned integer. UNSIGNED causes SORT to treat all numbers as positive. SIGNED is the default.

TRAILING_SIGN
LEADING_SIGN

Use this subqualifier with decimal data. It indicates whether the sign is stored at the beginning or end of the key. TRAILING_SIGN is the default.

OVERPUNCHED_SIGN
SEPARATE_SIGN

Use this subqualifier with decimal data. It indicates whether the sign is overpunched or is separated from the decimal. The default is OVERPUNCHED_SIGN.

For decimal data with a separate sign, ignore the sign byte in figuring size.

NUMBER=n

Use this subqualifier if you do not list your keys in order of descending priority. Here n specifies the priority of the key in the sort or merge operation. Acceptable values for n are 1–10, where 1 is the primary sort key, 2 is the secondary sort key, and so on.

If you do not specify NUMBER, SORT assumes NUMBER=1 for the first key you list, and the NUMBER value of the previous key plus 1 for each following key.

ASCENDING
DESCENDING)

This subqualifier indicates whether the key is to be sorted or merged in ascending or descending order. (Ascending order is A to Z or lowest to highest number.) ASCENDING is the default.

/PROCESS = **RECORD**
TAG
ADDRESS
INDEX

Use /PROCESS with SORT only. It indicates the type of sort to be performed. /PROCESS=RECORD is the default.

/STABLE

Use /STABLE with SORT only. It causes SORT to keep records with equal keys in the same order as in the input file. For multiple input files, records with equal keys in the first file will be placed before those from the second file, and so on.

If you use /STABLE, maximum key size is 251 instead of 255.

/CHECK_SEQUENCE

Use /CHECK_SEQUENCE with MERGE only. It causes MERGE to check all input files for correct sequence.

/COLLATING_SEQUENCE= **ASCII**
EBCDIC

This qualifier specifies the collating sequence to be used for character keys. COLLATING_SEQUENCE=EBCDIC causes ASCII characters to be sorted according to the EBCDIC sequence; characters remain ASCII representation. For MERGE, this qualifier specifies the sequence in which input records are arranged. COLLATING_SEQUENCE=ASCII is the default.

/WORK_FILES=n

Use /WORK_FILES with SORT only. Here n specifies the number of temporary work files to be used during the sort. Acceptable values are 0 and 2 through 10. The default number of work files is 2.

/STATISTICS

Use /STATISTICS with SORT only. It causes SORT to display a statistical summary at the end of your sort operation.

/SPECIFICATION[=file-specification]

Use /SPECIFICATION with SORT only. It causes SORT to refer to a specification file for SORT instructions. You can specify the name of the file which contains these instructions; if no file name is given, SYS$INPUT is assumed.

/RSX11

Use this qualifier to invoke SORT–11. /RSX11 requires PDP–11 SORT qualifiers with DCL syntax.

2.3.2 Input File Qualifiers

List the input file qualifier once following all (one to ten) input file specifications. The qualifier must apply to all input files.

/FORMAT=

(RECORD_SIZE=n

Use this subqualifier if record size is not to be obtained from RMS. Here n specifies the longest record length (LRL) in bytes of all input files. The longest record length allowed is 32,767 bytes (16,383 for relative files).

FILE_SIZE=n)

Use this subqualifier if file size is not to be obtained from RMS. Here n specifies the input file size in blocks. Maximum file size accepted is 4,294,967,295 blocks. For multiple input files, n is the total size of all files.

If file size is not provided by RMS or the user, SORT assumes 1000 blocks. SORT uses file size information to determine the size of its work files.

If you specify only one /FORMAT subqualifier, you can omit the parentheses.

2.3.3 Output File Qualifiers

List only one output file specification.

/FORMAT=

(FIXED=n
(VARIABLE=n
(CONTROLLED=n

Use this qualifier if you want your output file record format to differ from input file record format. You can specify fixed-length records (FIXED), variable-length records (VARIABLE), or variable with fixed-length control records (CONTROLLED). Default output record format is the same as input file record format if you are performing record or tag sort, and FIXED for address or index sort.

You can also specify with each format subqualifier the longest record length (LRL) of the output records (n). LRL is optional. The longest record length allowed is 32,767 bytes (16,383 bytes for relative files).

SIZE=n

Use this qualifier for CONTROLLED records only. Here n specifies the size in bytes of the fixed portion of CONTROLLED records. Maximum size is 255 bytes. Default size is that of the input file if it contains CONTROLLED records, or 2 bytes if input file record format is not CONTROLLED.

BLOCK_SIZE=n)

Use this qualifier with magnetic tape files only. Specify the block length of the output file in bytes. Default value is the block size established at tape mounting time or, if size was not specified, the block size of the input file tape. Block size must be 18 to 65,535 bytes.

NOTE

To assure correct data interchange with other DIGITAL systems, specify a block size of not more than 512 bytes. To assure compatibility with most non–DIGITAL systems, the block size should not exceed 2048 bytes.

/SEQUENTIAL
/RELATIVE
/INDEXED_SEQUENTIAL

These qualifiers specify output file organization. Default output file organization is the same as input file organization for record or tag sort, and /SEQUENTIAL for address or index sort.

If you specify /INDEXED_SEQUENTIAL (or if the output file is INDEXED_SEQUENTIAL by default), the output file must exist, its organization must be indexed sequential, and it must be empty; therefore, you must also specify /OVERLAY.

/ALLOCATION = n

This qualifier causes allocation of a certain number (n) of 512–byte blocks of disk space for your output file. Acceptable values are from 1 to 4,294,967,295.

/CONTIGUOUS

This qualifier indicates contiguous allocation of blocks for your output file. It must be used with /ALLOCATION.

/LOAD_FILL

Use this qualifier only with indexed files. It loads the buckets according to the fill size established when the file was created.

/OVERLAY

Use /OVERLAY to write the sorted or merged file into an existing empty file.

/BUCKET_SIZE = n

This qualifier specifies the number of 512–byte blocks per bucket for the output file. Default value is the same as the input file bucket size if input and output files have the same file organization; if they do not, default value is 1. The maximum number of blocks per bucket is 32.

2.4 Specification File

A specification file enables you to instruct the sort utility without listing qualifiers in a command line. Instead, you enter the qualifiers in a specification file and refer SORT to that file for instructions. Sorting with a specification file can prevent errors and save time in frequently performed sorting operations.

VAX–11 SORT accepts specification files created for SORT–11; however, it does not accept record type specifications or ALTSEQ records. That is, you

cannot omit records from your sort or specify an alternate collating sequence.

A specification file must contain two types of records:

- A Header Record, which tells the SORT program which sorting process you want and establishes either ascending or descending as "normal" sorting order
- One Field Specification Record for each key by which you want to sort. This record specifies data type, position, size and sorting order for that key

You can format these records in one of two ways:

- Fixed position field format: You enter data in fixed position fields based on card columns. This format permits conversion from SORT–11 specification files.
- Free field format: You enter data in the same order as fixed format but in variable-length fields separated by commas.

2.4.1 Header Record

The first record in a specification file must be the Header. It tells SORT the sorting process and order you want. You also enter information about key size and output record size here.

Enter instructions in the positions or order shown in Table 2–2. For free field format, enter a comma after each field, whether or not you enter data there. Notice that VAX–11 SORT does not use all available fields.

Table 2–2: Header Record

Field	Columns	Entry	Explanation
2	3–5	1	Identifies position of this record in your specification file Optional (for documentation purposes)
3	6	H	Identifies this record as Header Record
4	7–12	SORTR SORTT SORTA SORTI	Specifies the sorting process you want SORTR (record sort) is the default SORTT (tag sort), SORTA (address sort), and SORTI (index sort) are options
5	13–17	1–255	Specifies total of all key field sizes Optional for disk files
6	18	A or D	Establishes ascending (A) or descending (D) as "normal" sort order for keys Ascending is default
8	29–32	1–9999	Specifies largest output record length Optional for disk files
9	33–132	Comments	May be used for comments Precede free field entries by !

Notice that fields 1 and 7 (columns 1–2 and 19–28) are not used; SORT ignores any data in these columns. However, in free field format you must place commas after these fields.

In fixed field format, all numeric entries must be right-justified; either leading zeroes or blanks are acceptable.

2.4.2 Field Specification Record

Create one Field Specification Record for each key you plan to sort by. Enter them in your specification file in order of priority. Again, some fields are not used but need commas in free field format. Table 2–3 shows the order or position of fields.

Table 2–3: Field Specification Record

Field	Columns	Entry	Explanation
2	3–5	2–11	Identifies position of this record in your specification file Optional (for documentation purposes)
3	6	F	Identifies record as Field Specification
4	7	N or O	Specifies sorting order for this key as "Normal" or "Opposite" the order established in field 6 of the Header
5	8		Specifies code for data type
		C	C = ASCII Character (the default)
		B	B = Binary
		F	F = Single or double floating point
		D	D = Decimal with sign trailing and overpunched
		I	I = Decimal with sign leading and separate
		J	J = Decimal with sign trailing and separate
		K	K = Decimal with sign leading and overpunched
		P	P = Packed Decimal
		Z	Z = Zoned
6	9–12	1–32737	Position of first byte of key field Omit for one-byte keys
7	13–16	1–32737	Position of last byte of key field
9	20–80	Comments	May be used for comments Precede free field entries by !

Notice that fields 1 and 8 (columns 1–2 and 17–19) are not used; any data in these columns is ignored. However, in free field format you must place commas after these fields.

NOTE

VAX–11 SORT does not support G or H floating-point, EBCDIC, or unsigned data in a specification file sort.

Figure 2–5 shows specification files in both formats created for the reading-score sort done earlier in this chapter (Figure 2–2).

Figure 2–5: Sample Specification Files

Fixed Field Position Specification File

```
Column
     5     10    15  20        25   30

1H SORTR    19A                 80Header Record
2FNC00000011    Field Specification for grade
3FNC00140028    Field Specification for teacher
4FOC00540056    Field Specification for score
```

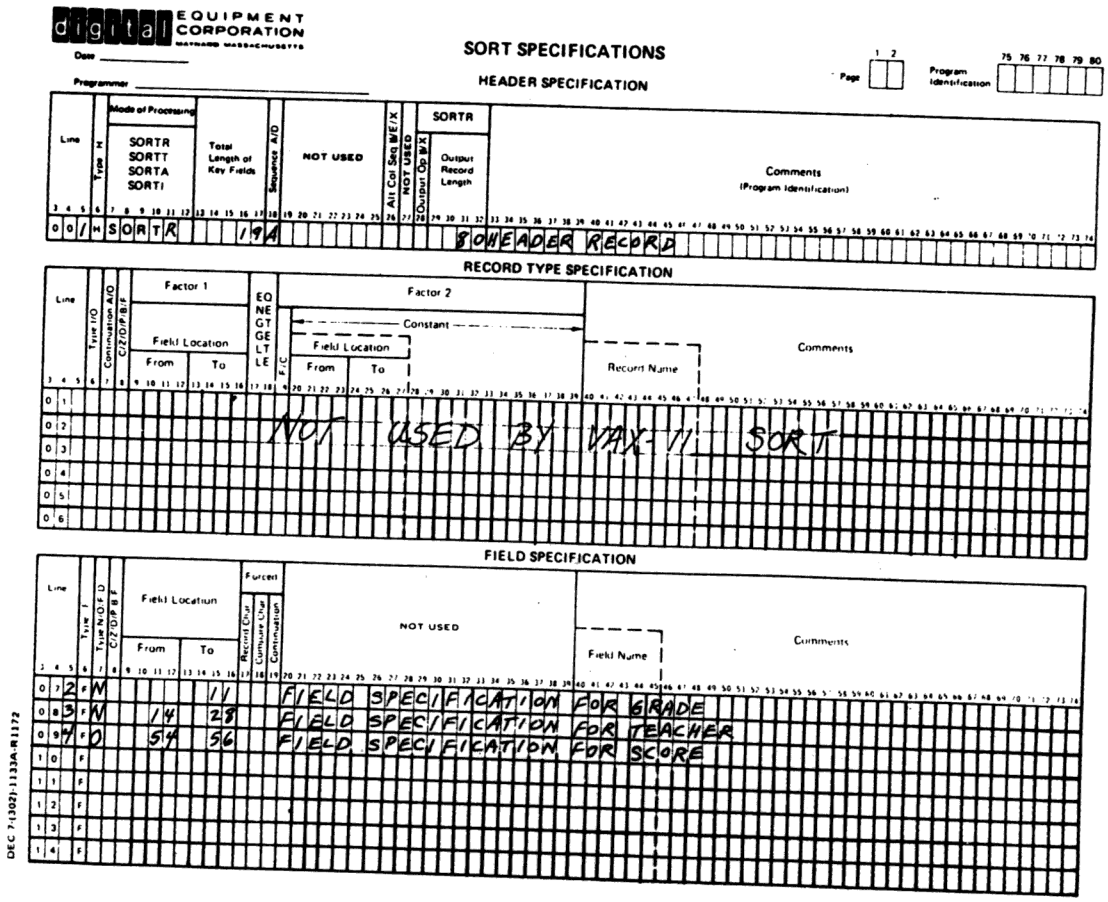
Free Field Specification File

```
,1,H,SORTR,19,A,,80,!Header Record
,2,F,N,C,,11,,!Field Specification for grade
,3,F,N,C,14,28,,!Field Specification for teacher
,4,F,O,C,54,56,,!Field Specification for score
```

For purposes of illustration, values available by default are filled in.

Digital Equipment Corporation provides a form, "Sort Specifications," which helps you select and arrange the data you need to create a specification file. Figure 2–6 shows this form.

Figure 2–6: SORT Specifications



2.4.3 Specification File Commands

You can enter specification file instructions by writing the Header Record and Field Specification Record(s) into a specification file.

If you have created a specification file in this way, type the SORT command as follows, placing the file specifications for your specification file (SPEC.FIL), input file (INPUT.FIL), and output file (OUTPUT.FIL) in the indicated positions:

```
$ SORT/SPECIFICATION=SPEC.FIL INPUT.FIL OUTPUT.FIL RET
```

SORT then executes.

You can also enter specification file instructions at the terminal when you invoke the SORT utility.

Instead of creating a specification file, you can type:

```
$ SORT/SPECIFICATION INPUT.FIL OUTPUT.FIL RET
```

DCL prompts with:

```
PLEASE ENTER SPECIFICATION FILE RECORDS
```

You can then enter the data for the Header Record on the first line, press return, and enter Field Specification data for the primary key on the next line. Continue entering key data on each following line until you have entered specifications for all keys. Then type CTRL/Z to indicate completion of your entries, and SORT executes. You can enter data in either free or fixed format. See Figure 2–7 for an example of prompted specification file instructions.

Figure 2–7: Prompted Specification File

```
$ sort/specification read.fil read.lis RET

PLEASE ENTER SPECIFICATION FILE RECORDS.

,1,H,SORTR,19,A,,80,
,2,F,N,C,,11,,
,3,F,N,C,14,28,,
,4,F,0,C,54,56,,
^Z
```

If you type only:

```
$ SORT/SPECIFICATION RET
```

SORT first prompts for input and output files, then requests specification file records.

Chapter 3
Calling SORT and MERGE from User Programs

Often your sort or merge task is part of a larger operation you are performing on your files. Perhaps you want to process the records in some way before you sort them, or after they are sorted and before you print or store them. Or you may want to select certain records from your files, merge them, then print or display the merged file. In such cases, you can call sort/merge subroutines from a program you write to perform several tasks.

Eight subroutines, each of which executes a portion of the sort or merge operation, make up the sort/merge utility program. When you use the command, you do not see the separate subroutines. However, when you include these subroutines in a program, you call them one at a time.

You can call the sort/merge subroutines from any VAX–11 native mode language. These include:

VAX–11 BASIC

VAX–11 BLISS–32

VAX–11 COBOL

VAX–11 COBOL–74

VAX–11 CORAL–66

VAX–11 FORTRAN

VAX–11 MACRO

VAX–11 PASCAL

You use the same calls from all languages. Calls and associated parameters conform to the VAX–11 Calling Standard. (See the *VAX–11 Architecture Handbook* for an explanation of the Calling Standard.) VAX–11 SORT uses reference parameters and descriptor parameters for passing data. Because different programming languages express these types of parameters differently, this chapter does not provide instructions for each language. For help, refer to the reference manual or user's guide for the VAX–11 programming language in which you are writing your program. Unless noted otherwise, all parameters described in this chapter are passed by reference.

Because the subroutines are part of the VAX/VMS Common Run-Time Library, they are automatically included in the image when you link.

This chapter:

- Provides an overview of SORT and MERGE
- Describes the sort and merge subroutines and their mandatory and optional parameters
- Presents sample programs that call the subroutines from native-mode languages

3.1 Two I/O Interfaces

You can submit your records for sorting or merging as complete file(s) or as single records. These alternatives are called I/O interfaces.

In file interface your program submits one or more files to SORT or MERGE, which creates one sorted or merged output file. In record interface your program submits records to the utility one at a time, then receives the sorted or merged records one at a time.

File interface executes faster than record interface and is easier to incorporate into your program. If you use it, you must sort (or merge) all records in the file(s) and cannot process records before or after sorting. The record interface permits you to perform an operation on each record before or after sorting or merging.

If, for example, you want to eliminate duplicate records before merging several files, use the record interface.

The set of calls differs for the two interfaces. After deciding which interface you want, select the necessary calls. Section 3.2 describes the sort subroutines; Section 3.3 describes the merge subroutines. Figure 3–3 provides a summary of subroutines and parameters.

3.2 SORT Subroutines

As with the SORT command, when you call the sort subroutines you must provide:

- Information about keys (for example, data type, size, and position)
- File specifications of input and output files (for file interface)
- Instructions about sort method (for example, ascending or descending; record, tag, address or index sort)

You pass this information to SORT by means of the subroutine parameters.

After being called, each subroutine performs its function and returns control to your program. It also returns a 32–bit status code indicating success or error in that phase of the sort. Your program can test that value to determine success or failure conditions. Programs in most VAX–11 native mode languages must declare the subroutines functions in order to do this testing.

When you call sort subroutines from a program, SORT does not provide a statistical summary of the operation, as it can when you use the SORT command.

Table 3–1 shows the order of calls for each interface and briefly indicates the function of each. Sections 3.2.1 to 3.2.6 describe them in detail.

Table 3–1: SORT Subroutines

File Interface	
SOR$PASS_FILES	Passes names of input and output files to SORT; must be repeated for multiple input files
SOR$INIT_SORT	Passes key information and SORT options
SOR$SORT_MERGE	Sorts the records
SOR$END_SORT	Does housekeeping functions such as closing files and releasing memory
Record Interface	
SOR$INIT_SORT	Passes key information and SORT options
SOR$RELEASE_REC	Passes one input record to the sort; must be called once for each record
SOR$SORT_MERGE	Sorts the records
SOR$RETURN_REC	Returns one sorted record to your program; must be called once for each record
SOR$END_SORT	Does housekeeping functions such as closing files and releasing memory

File Interface

For a sort using file interface, your first step is to pass the input and output file specifications to SORT. Call SOR$PASS_FILES and use descriptor parameters to pass the file specifications.

Next, call SOR$INIT_SORT, using its parameters to pass your instructions about keys and sort options. Sorting is essentially comparing one key with another over and over. You must indicate at this point whether you want SORT to do key comparison for you or to use a key comparison routine you provide.

SORT automatically generates the key comparison routine that is most efficient for your key data type(s). However, you may want to provide your own comparison routine to handle special sorting requirements. For example, you may want a group of records to be placed at the beginning of the output file. Or you may want names beginning with "Mc" and "Mac" to be placed together.

Provide SOR$INIT_SORT with a parameter referring it to either the key buffer or your key comparison routine. The key buffer contains information SORT needs to do the compares, such as number of keys, data type, position, length, and sort order.

Next, call SOR$SORT_MERGE to execute the sort. It takes no parameters. Finally, call SOR$END_SORT to end the sort and release memory.

Record Interface

For record interface the first call is SOR$INIT_SORT. As in file interface, its parameters define keys and sort options.

Next, call SOR$RELEASE_REC, passing it a parameter that refers SORT to the record buffer in your data area. Your program must call SOR$RELEASE_REC once for each record to be released to SORT.

After your program has passed all records to SORT, it must call SOR$SORT_MERGE to perform the sorting.

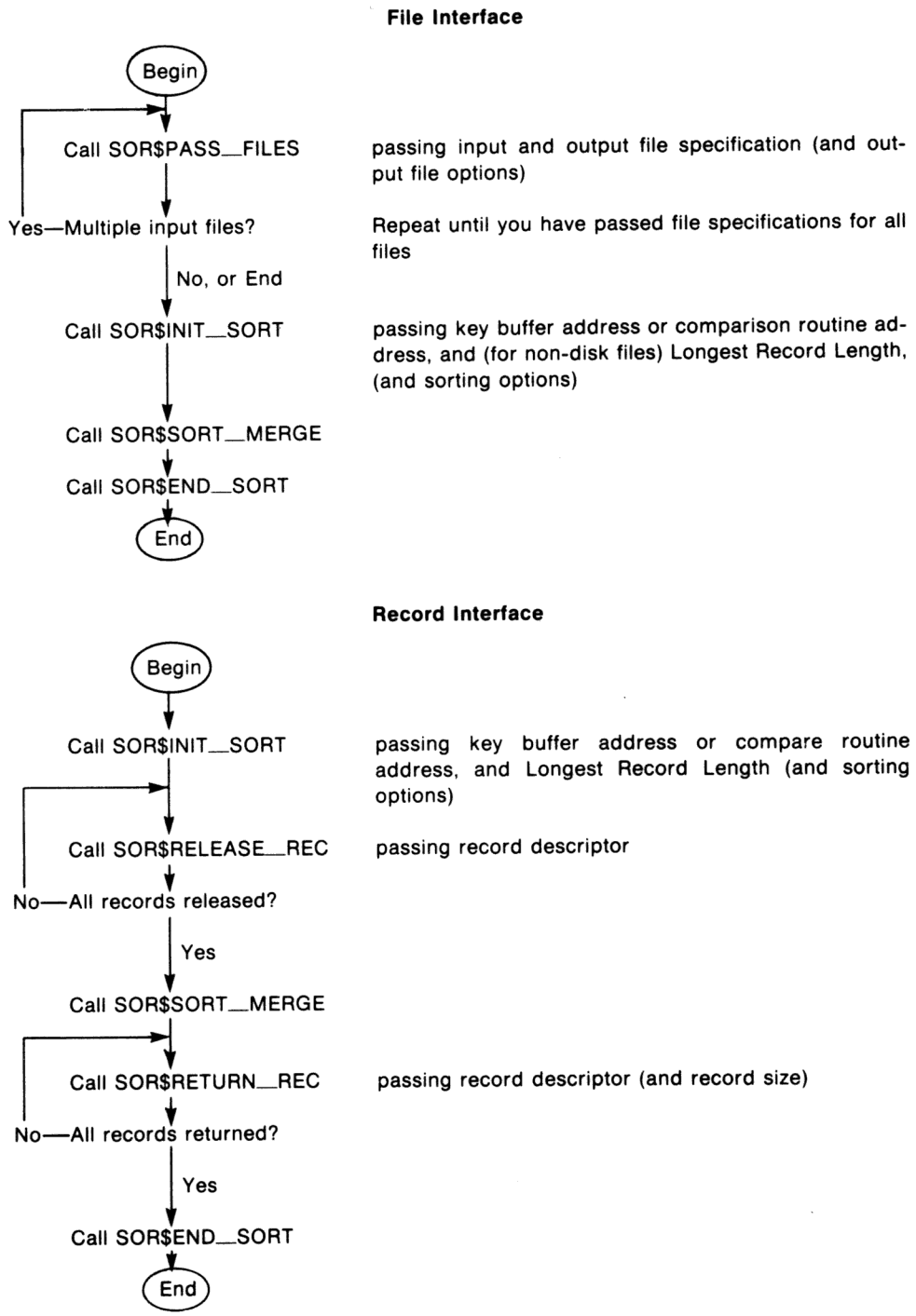
Next, you must ask SORT to return the sorted records to your program. Do this by calling SOR$RETURN_REC once for each record to be returned. A parameter tells SORT where to place the record.

Finally, call SOR$END_SORT to complete the sort.

Sections 3.2.1 through 3.2.6 describe each sort subroutine in detail, including its mandatory and optional parameters and the error codes it returns.

For quick reference, Figure 3–1 gives a flowchart summary of calls and parameters.

Figure 3–1: Flowchart for SORT Subroutines



3.2.1 SOR$PASS_FILES

SOR$PASS_FILES uses two mandatory parameters to pass the names of input and output files to SORT. SORT needs these to open input file(s) and create an output file for the sorted records. Only file interface calls this subroutine. MERGE file interface also calls SOR$PASS_FILES.

This subroutine can also accept up to seven optional parameters that provide specifications for the output file, such as file organization, format, and size. If you do not want default values, you can specify other values. These options are like the output file qualifiers of the SORT command (see Section 2.1.3).

The two mandatory parameters are descriptor parameters:

1. **Input file descriptor(s)**

2. **Output file descriptor**

   When you call SOR$PASS_FILES, you must pass to SORT the file specifications of input and output files. For multiple input files, call SOR$PASS_FILES once for each input file, passing one input file descriptor each time. Specify the output file descriptor only once, as part of the first call.

   For example:
   ```
   Call SOR$PASS_FILES(INPUT1,OUTPUT)
   Call SOR$PASS_FILES(INPUT2)
   Call SOR$PASS_FILES(INPUT3)
   Call SOR$PASS_FILES(INPUT4)
   ```

   Each file descriptor is a VAX–11 standard string descriptor. It is composed of two longwords that describe the ASCII string that is the file specification of the input or output file. The first word of the first longword contains the length of the ASCII string (SORT does not use the second word of the first longword). The second longword contains a name that is the address of the string in your program's data area.

   Some languages, such as FORTRAN, create the descriptor for you. Others, such as MACRO, do not. In the sample MACRO program in Section 3.9, the name FILEIN is used with the call and refers SORT to the input file descriptor. The file descriptor contains the length (10) and the address (FILENAMEIN). These refer in turn to the input file specification, "R010SQ.DAT".

   In the sample FORTRAN program in Section 3.7, OUTPUTNAME is a parameter of SOR$PASS_FILES. At other places in the program, OUTPUTNAME is defined as 8 ASCII characters and is associated with the file specification "TEST.TMP". Because FORTRAN passes CHARACTER quantities by descriptor by default, simply specifying OUTPUTNAME in the argument list is the correct way to call SOR$PASS_FILES.

The seven reference parameters that follow are optional.

3. **Output file organization**

   Specify a byte containing one of the following values:

   ```
   FAB$C_SEQ
   FAB$C_REL
   FAB$C_IDX
   ```

   Default value is file organization of the first input file for record or tag sort and SEQ for address and index sort.

See the $FAB ORG parameter in the *VAX–11 Record Management Services Reference Manual* for information on these options.

4. **Output file record format**

   Specify a byte containing one of the following values:

   FAB$C_FIX
   FAB$C_VAR
   FAB$C_VFC

   Default value is record format of the first input file.

   See the $FAB RFM parameters in the *VAX–11 Record Management Services Reference Manual* for information on these options.

5. **Output file bucket size**

   Specify a byte containing the bucket size. Acceptable values are 1 to 32 blocks.

   Default value is the bucket size of the first input file if the output file organization is the same as the input file organization. If the file organizations are different, default value is 1 block.

6. **Output file block size**

   Specify an unsigned word containing the block size. Acceptable values are 18 to 65,535.

   Use this parameter with magnetic tape files only. Default value is the block size established when the file is mounted or, if no size is specified, the block size of the input file tape.

7. **Output file maximum record size**

   Specify a word containing the maximum record size for the output file. Acceptable values are 0 to 32,767 (0 to 16,383 for relative files).

   The value 0 means you want no maximum size check; 1 to 32,767 limits record size to the specified value.

   See the $FAB MRS parameter in the *VAX–11 Record Management Services Reference Manual* for more information.

8. **Output file allocation**

   Specify a longword containing the number of blocks you want to allocate to the output file. Acceptable values are 1 to 4,294,967,295.

   Default value is the number of blocks needed for the output file, based on the total input file allocation.

9. **Output file options**

   Specify a longword containing the options you want to place in the FAB$L_FOP field.

   See the *VAX–11 Record Management Services Reference Manual* for an explanation of FAB$L_FOP file options.

Separate parameters with commas. Also, enter a comma for any parameter you skip. However, you can truncate the parameter list at any point.

Table 3–2 shows the status codes SORT can return to your program after completing this phase of the sort.

Table 3–2: SOR$PASS_FILES: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_INCONSIS	1C805C	1867868	Inconsistent data for file
SOR$_OPENIN	1C109C	1839260	Cannot open input file
SOR$_OPENOUT	1C10A4	1839268	Cannot open output file
SOR$_SORT_ON	1C802C	1867820	A sort is in progress or this call is in the wrong sequence
SOR$_VAR_FIX	1C8064	1867876	Cannot change variable records to fixed records
SS$_NORMAL	1	1	Success

3.2.2 SOR$INIT_SORT

This subroutine sets up sort work areas and passes parameters that provide key specifications and sort options. Both interfaces call it.

SOR$INIT_SORT accepts eight reference parameters. You must use either the key buffer address parameter or the comparison routine address parameter. In addition, for record interface or files not on disk, you must use the longest record length parameter. Five other parameters are optional: input file size, number of work files, sort type, total key size, and sort options.

1. **Key buffer address**

   If you want SORT to do key compares, you must pass the key buffer address to SORT. Specify the address of the key buffer in your data area.

   In the key buffer, you define the keys by which you plan to sort. You must provide information about:

   - Number of keys
   - Data type
   - Ascending or descending sort
   - Start position
   - Length

The key buffer is an array of words describing the keys. The first word contains the number of keys. Following this word is a block of four words for each key. Each block contains:

- Key type — one word giving code for data type (see Table 3–3)
- Key order — one word, 0 or 1, indicating ascending (0) or descending (1) sort
- Start position — one word indicating key position (1 to record size)
- Length — one word indicating key length (1–255)

You can sort by up to ten key fields. Specify keys in the desired sorting order, with primary key first and most minor key last.

Table 3–3: Data Type Codes

1 =	Character
2 =	Binary
3 =	Zoned
4 =	Packed decimal
5 =	Unsigned binary
6 =	Decimal leading overpunched
7 =	Decimal leading separate
8 =	Decimal trailing overpunched
9 =	Decimal trailing separate
10 =	F format floating point
11 =	D format floating point
12 =	G format floating point
13 =	H format floating point

For example, suppose that you have a file of records containing a 40–character name field beginning at position 10 and a 10–digit packed decimal field for transaction number beginning at position 60. If you want to sort by name as primary key, and transaction number as secondary key, set up your key buffer this way:

```
          2 = number of keys

Key 1     1 = key type (character)
          0 = key order (ascending)
         10 = start position in record
         40 = length of key

Key 2     4 = key type (packed-decimal)
          0 = key order (ascending)
         60 = start position in record
         10 = length of key in digits
```

If the key buffer specifies the number of keys as zero, SORT assumes that no key buffer has been passed.

You do not need this parameter if you provide your own key comparison routine. If you pass both key buffer address and comparison routine address, SORT ignores the comparison routine address.

NOTE

For record interface, you cannot sort decimal data types other than packed decimal unless you provide your own key comparison routine. However, you can convert any decimal format to packed decimal using the decimal instructions in the VAX–11 instruction set.

2. **Longest record length** (LRL)

   Specify a word containing the size of the longest record in the file.

   This parameter is mandatory for record interface and input files not on disk.

3. **Input file size**

   Specify a longword containing the input file size in blocks.

   This parameter can increase SORT efficiency in record interface and in sorting files not on disk.

4. **Number of work files**

   Specify a byte containing the number of work files you want SORT to use in the sorting process. Acceptable values are 0 and 2–10. Default value is 2.

5. **Sort type**

   Specify a byte indicating the type of sort you want.

   1 = Record sort
   2 = Tag sort
   3 = Index sort
   4 = Address sort

   Only file interface using a single input file accepts this parameter. If you select record interface or sort multiple files, you can perform only a record sort. Default is record sort.

6. **Total key size**

   Specify a byte giving total key size. Acceptable values are 1–255.

7. **Comparison routine address**

   Specify the address of your key comparison routine. SORT calls this routine with two reference parameters, the addresses of the two records you must compare. The routine must return a 32–bit integer value:

   –1 if key 1 should come before key 2

   0 if keys are equal

   1 if key 1 should come after key 2

The second FORTRAN program (Section 3.8), for example, uses KOMPAR as the address of its comparison routine, then passes the addresses of the two records in this way:

```
FUNCTION KOMPAR (REC1,REC2)
```

This parameter is mandatory if you do not use key buffer address.

Note that you cannot specify your own key comparison routine from VAX–11 COBOL–74 or VAX–11 COBOL.

8. **Sort options**

Specify a longword of bit flags containing the options you want to use in the sort. Values available in VAX–11 SORT are:

SOR$V_STABLE	keeps records with equal keys in order.
SOR$V_EBCDIC	orders ASCII character keys in EBCDIC collating sequence. No translation takes place.

All other bits in the longword are reserved and must be zero.

Table 3–4 shows the status codes SORT can return to your program after completing this phase of the sort.

Table 3–4: SOR$INIT_SORT: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_BAD_FILE	1C808C	1867916	An invalid file size
SOR$_BAD_LRL	1C8084	1867908	An invalid LRL was specified
SOR$_BAD_TYPE	1C806C	1867884	An invalid sort process was specified
SOR$_KEY_LEN	1C80AC	1867948	Invalid key length specified
SOR$_LRL_MISS	1C8074	1867892	No LRL was specified and is required
SOR$_MISS_KEY	1C8004	1867780	No key definition specified
SOR$_NUM_KEY	1C803C	1867836	Invalid number of keys specified (must be 1–10)
SOR$_SORT_ON	1C802C	1867820	A sort is in progress or this call is in the wrong sequence
SOR$_VM_FAIL	1C801C	1867804	SORT failed to get needed virtual memory
SOR$_WORK_DEV	1C800C	1867788	Work file device not random access device or not local node
SOR$_WS_FAIL	1C8024	1867812	SORT failed to get needed working set size
SS$_NORMAL	1	1	Success

3.2.3 SOR$RELEASE_REC

In record interface, this subroutine passes a record to SORT. Set up a record buffer in your program's data area to contain the record. Call SOR$RELEASE_REC once for each record to be sorted.

One descriptor parameter, record descriptor, is mandatory for this call.

Record descriptor

When you call SOR$RELEASE_REC, you pass a record to SORT. Use a string descriptor that describes your record buffer by its length and its address in your program's data area. Your language may create the descriptor for you.

The record buffer takes a different form depending on whether SORT is doing key compares or you are providing your own key comparison routine. If you are relying on SORT to do key compares, your record buffer is in fact a key-and-record buffer. That is, the record you pass to the SOR$RELEASE_REC subroutine consists of the sort keys followed by the record itself (including the keys). Specify keys in the same sequence and in the same way (for example, the same size and data type) that they appear in the key buffer. Leave no space between key fields or between the key area and the following record area.

Thus, for sort compares, record length is equal to the total length of the record plus the total length of the key fields. The address is the first byte of the first key.

However, if you are using your own comparison routine, the buffer contains only the record itself. The descriptor length is the record length and the buffer address is the first byte of the record.

Table 3–5 shows the status codes SORT can return to your program after this phase of the sort.

Table 3–5: SOR$RELEASE_REC: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_BAD_ADR	1C8094	1867924	Invalid descriptor address passed
SOR$_BAD_LRL	1C8084	1867908	Record length is longer than LRL specified
SOR$_EXTEND	1C80A4	1867940	Failed to extend work file
SOR$_KEY_LEN	1C80AC	1867948	Invalid key length specified
SOR$_MAP	1C809C	1867932	Internal sort map error
SOR$_NO_WRK	1C8014	1867796	Cannot do sort in memory, need work files
SOR$_SORT_ON	1C802C	1867820	A sort is in progress or this call is in the wrong sequence
SS$_NORMAL	1	1	Success

3.2.4 SOR$SORT_MERGE

This subroutine sorts the file. Call it once after you have passed file names or records to SORT and after SOR$INIT_SORT has set up key specifications. It takes no parameters.

Table 3–6 shows status codes returned to your program after this phase of the sort.

Table 3–6: SOR$SORT_MERGE: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_BADFIELD	1C101C	1839132	Bad data in key field
SOR$_EXTEND	1C80A4	1867940	Failed to extend work file
SOR$_MAP	1C809C	1867932	Internal sort map error
SOR$_NO_WRK	1C8014	1867796	Cannot do sort in memory, need work files
SOR$_READERR	1C10B4	1839236	Cannot read a specified input file record
SOR$_SORT_ON	1C802C	1867820	A sort is in progress or this call is in the wrong sequence
SOR$_WRITEERR	1C10D4	1839316	Cannot write a specified output file record
SS$_NORMAL	1	1	Success

3.2.5 SOR$RETURN_REC

This subroutine returns one sorted record to the user program. It places the record into a record buffer that you specify. It returns the record length as well. MERGE record interface also calls SOR$RETURN_REC. Call this subroutine once for each record sorted or merged.

SOR$RETURN_REC requires one descriptor parameter, record descriptor, and one reference parameter, record size.

The subroutine returns the SS$_ENDOFFILE status if there are no more records to return.

1. **Record descriptor**

   This is a string descriptor that describes the record buffer that receives the sorted record. In sorting, this buffer can be the same one from which the record was released, but in merging it cannot. SOR$RETURN_REC also accepts a dynamic string descriptor for this parameter.

   Before SORT returns the record, it strips off the leading keys from the key-and-record configuration used for SORT key compares. The length removed is the value specified as total key size to SOR$INIT_SORT. The returned record, therefore, is equal to either (1) the record passed with the user compare routine or (2) the record portion of the key-and-

record buffer used for SORT key compares. The length you specify in the record descriptor is the length of your record only. The address is that of the first byte of the record.

2. **Record size**

   Specify a word that receives the actual length of the returned record.

Table 3–7 shows status codes returned to your program at the end of this phase of the sort.

Table 3–7: SOR$RETURN_REC: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_BAD_LRL	1C8084	1867908	Record length is longer than LRL specified
SOR$_BAD_ORDER	1C80D4	1867988	Input file UL is out of order
SOR$_EXTEND	1C80A4	1867940	Failed to extend work file
SOR$_MAP	1C809C	1867932	Internal sort map error
SS$_ENDOFFILE	870	2160	Success, no more records to return
SS$_NORMAL	1	1	Success, a record has been returned

3.2.6 SOR$END_SORT

This subroutine closes files, cleans up sort work areas, and releases memory. It takes no parameters. Call SOR$END_SORT once to end the sort. You cannot begin another sort operation before calling this subroutine.

Table 3–8 shows status codes returned at the completion of the sort.

Table 3–8: SOR$END_SORT: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_CLEANUP	1C80B4	1867908	Failed to delete work files and reinitialize work areas and data areas
SS$_NORMAL	1	1	Success

3.3 MERGE Subroutines

When you call the merge subroutines, you must provide:

- Number of input files
- File specifications of input and output files (for file interface)
- Information about merge keys
- Input routine (for record interface)

You pass this information to MERGE by means of the subroutine parameters.

After performing its function, each subroutine returns to your program a 32–bit status code indicating success or error in that phase of the merge. Your program can then test that value to determine success and failure conditions. Programs in most VAX–11 native mode languages must declare the subroutines' functions in order to do this testing.

Table 3–9 shows the order of calls for each interface and briefly indicates the function of each. Since sorting involves merging, MERGE uses some of the SORT subroutines. It also uses two other subroutines, described in Sections 3.3.1 and 3.3.2.

Table 3–9: MERGE Subroutines

File Interface	
SOR$PASS_FILES	Passes names of input and output files to MERGE; called once for each input file
SOR$INIT_MERGE	Sets up key parameters
SOR$DO_MERGE	Does the merge and cleans up
Record Interface	
SOR$INIT_MERGE	Sets up key parameters and calls input routine
SOR$RETURN_REC	Returns merged records and continues to call input routine; called once for each record
SOR$END_SORT	Cleans up

File Interface

For a merge using file interface, your first step is to pass the input and output file specifications to MERGE. Call SOR$PASS_FILES and provide it with the file specifications.

You can merge from 2 to 10 input files; call SOR$PASS_FILES once for each file. Pass the file specification for the one merged output file in the first call.

Next, call SORT$INIT_MERGE, using its parameters to pass your instructions about keys and merge options. Just as in sorting, merging involves comparing one key with another over and over. You can allow MERGE to do key comparison for you or provide your own comparison routine tailored to your data. Provide a parameter referring SOR$INIT_MERGE to either your key buffer or your key comparison routine.

Next, call SOR$DO_MERGE. This subroutine executes the merge and cleans up.

Record Interface

For record interface, first call SOR$INIT_MERGE. As in file interface, its parameters define keys and merge options. It also issues the first call to your input routine, which begins releasing records to the merge.

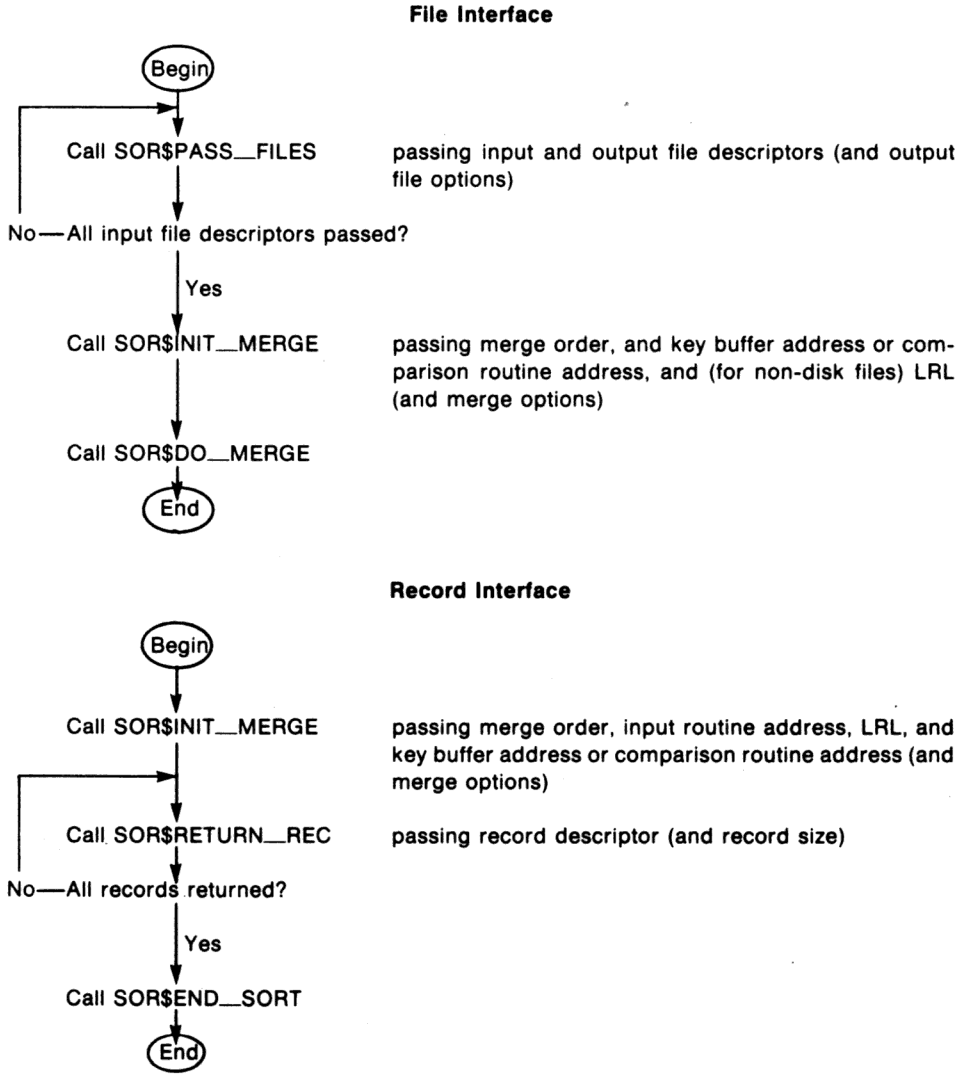
Next, call SOR$RETURN_REC. It returns a merged record and continues to call the input routine. In merging, unlike sorting, MERGE does not need to hold all records before it can begin returning them in the desired order. Releasing, merging, and returning of records are all taking place in this phase of the merge.

Call SOR$RETURN_REC until all records have been returned. A parameter tells MERGE where to place the merged record.

Finally, call SOR$END_SORT to clean up.

Sections 3.3.1 and 3.3.2 describe the merge subroutines. For quick reference, Figure 3–2 gives a flowchart summary of calls and parameters.

Figure 3–2: Flowchart for MERGE Subroutines



3.3.1 SOR$INIT_MERGE

This subroutine accepts parameters that provide key specifications and merge options and tell MERGE the number of input files. Both interfaces call it.

SOR$INIT_MERGE accepts six reference parameters. You must always specify merge order. In addition, you must specify either key buffer address or comparison routine address. For record interface, you must also pass input routine address and longest record length. Finally, for files not on disk, you must pass longest record length. An optional parameter contains merge options.

1. **Merge order**

   Specify a byte containing the number of input files (2–10). Merge order is a mandatory parameter in all merge operations.

2. **Key buffer address**

   Specify the address of your key buffer array. Set up the key buffer array in the same way as for SOR$INIT_SORT.

   This parameter is mandatory unless you are using comparison routine address.

3. **Longest record length** (LRL)

   Specify a word containing the size of the longest record in any of the input files.

   This parameter is mandatory for record interface and for non-disk files.

4. **Merge options**

   Specify a longword of bit flags containing the options you want for the merge. Values available are:

SOR$V_EBCDIC	Merges ASCII character keys in the EBCDIC collating sequence. Does not translate characters.
SOR$V_SEQ_CHECK	Requests an "out of order" error return if an input file is not in correct order.

   All other bits in the longword are reserved and must be zero.

5. **Comparison routine address**

   Specify the address of your key comparison routine. Set up this routine in the same way as for SORT$INIT_SORT.

   This parameter is mandatory if you do not use key buffer address.

6. **Input routine address**

   Specify the address of a routine you write to release a record to the merge. SOR$INIT_MERGE and SOR$RETURN_REC call this routine until all records have been passed.

Your routine must read (or construct) a record, place it in a record buffer, store its length in an output parameter, then return control. MERGE compares keys and returns records in merged order until it has processed all records. The input routine must accept three parameters and return a status value. The parameters provide for receiving and storing information necessary to the merge operation, such as the next file from which to read a record. Use the following parameters with your input routine.

- A descriptor for the buffer where your routine must place the record.
- A longword (passed by reference) containing the file number from which to input a record. The first file is 1, the second 2, and so on.
- A word (passed by reference) where your routine must return the actual length of the record.

Your routine must return a status value. If you return SS$_NORMAL (or any other success status), the merge continues, using the record that you released. If you return SS$_ENDOFFILE, MERGE notes that you have no more records for the file specified by parameter 2, and ignores the contents of the record buffer. If you return any other error status, MERGE terminates and passes the status value back to the caller of SOR$INIT_MERGE or SOR$RETURN_REC.

The second FORTRAN program (Section 3.8), for example, specifies an input routine in this way:

```
FUNCTION READ_REC (RECX, FILE, SIZE)
```

You must use this parameter in a merge using record interface.

Table 3–10 shows the status codes returned to your program after this phase of the merge.

Table 3–10: SOR$INIT_MERGE: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_BAD_KEY	1C8034	1867828	Invalid key specification
SOR$_BAD_MERGE	1C80BC	1867964	Merge order must be between 2 and 10
SOR$_KEY_LEN	1C80AC	1867948	Invalid key length specified
SOR$_LRL_MISS	1C8074	1867892	No LRL was specified and is required
SOR$_MISS_KEY	1C8004	1867780	No key definition specified
SOR$_NUM_KEY	1C803C	1867836	Invalid number of keys specified (must be 1–10)

(continued on next page)

Table 3–10: SOR$INIT_MERGE: Possible Returns (Cont.)

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_SORT_ON	1C802C	1867820	A sort or merge is in progress or this call is in the wrong sequence
SOR$_VM_FAIL	1C801C	1867804	MERGE failed to get needed virtual memory
SS$_NORMAL	1	1	Success

3.3.2 SOR$DO_MERGE

This subroutine performs the merge and cleans up. It takes no parameters.

Table 3–11 shows the status codes returned to your program at the end of the merge.

Table 3–11: SOR$DO_MERGE: Possible Returns

Symbolic	Hex Value	Decimal Value	Meaning
SOR$_CLEAN_UP	1C80B4	1867908	Failed to delete work files and reinitialize work areas and data areas
SOR$_READERR	1C1084	1839236	Cannot read a specified input file record
SOR$_WRITEERR	1C10D4	1839316	Cannot write a specified output file record
SS$_NORMAL	1	1	Success

Figure 3–3 provides a summary of sort and merge subroutines and parameters. It indicates the place of each subroutine in sort/merge and in record/file interface, and it labels each parameter as reference or descriptor and as mandatory or optional.

Sections 3.4 to 3.11 contain sample programs calling the sort and merge subroutines from VAX–11 native mode languages.

In VAX–11 COBOL, you can execute a sort or merge with the SORT or MERGE statement, so no sample COBOL program is given here. You can, however, use the calls described in this chapter with VAX–11 COBOL and COBOL–74, and sort programs already written in COBOL–74 can be moved to VAX–11 COBOL. See Chapter 11 in the *VAX–11 COBOL User's Guide* for an explanation of using the COBOL SORT and MERGE verbs.

Figure 3–3: Subroutine Summary

Call Order SORT FILE	Call Order SORT REC	Call Order MERGE FILE	Call Order MERGE REC	Subroutine	Parameter	Mandatory/ Optional	Reference/ Descriptor
1		1		SOR$PASS_FILES	1. Input file descriptor	M	D
					2. Output file descriptor	M	D
					3. Output file organization	O	R
					4. OP file record format	O	R
					5. OP file bucket size	O	R
					6. OP file block size	O	R
					7. OP file max record size	O	R
					8. OP file allocation	O	R
					9. OP file options	O	R
2	1			SOR$INIT_SORT	1. Key buffer address	M:1or7	R
					2. LRL	M for record & non-disk files	R
					3. Input file size	O	R
					4. Number of work files	O	R
					5. Sort type	O	R
					6. Total key size	O	R
					7. Comparison routine address	M:1or7	R
					8. Sort options	O	R
	2			SOR$RELEASE_REC	1. Record descriptor	M	D
3	3			SOR$SORT_MERGE	None		
	4		2	SOR$RETURN_REC	1. Record descriptor	M	D
					2. Record size	M	R
4	5		3	SOR$END_SORT	None		
		2	1	SOR$INIT_MERGE	1. Merge order	M	R
					2. Key buffer address	M:2or5	R
					3. LRL	M for record & non-disk files	R
					4. Merge options	O	R
					5. Comparison routine address	M:2or5	R
					6. Input routine address	M:record	R
		3		SOR$DO_MERGE	None		

3.4 Sample VAX–11 BASIC Program (SORT)

```
100         EXTERNAL INTEGER FUNCTION SOR$INIT_SORT                 &
\           EXTERNAL INTEGER FUNCTION SOR$RELEASE_REC               &
\           EXTERNAL INTEGER FUNCTION SOR$SORT_MERGE                &
\           EXTERNAL INTEGER FUNCTION SOR$RETURN_REC                &
\           EXTERNAL INTEGER FUNCTION SOR$END_SORT                  &
\           EXTERNAL INTEGER CONSTANT SS$_ENDOFFILE

200         MAP (KEYB) KEY$=8, REC$=32, WORD KEY_BUF(4)             &
\           MAP (KEYB) KEY_AND_REC$=40

250         KEY_BUF(0%) = 1%    ! One key                           &
\           KEY_BUF(1%) = 1%    ! Character key type               &
\           KEY_BUF(2%) = 0%    ! Ascending order                  &
\           KEY_BUF(3%) = 1%    ! Starting position                &
\           KEY_BUF(4%) = 8%    ! Size of key

300         SORT_STATUS% = SOR$INIT_SORT(KEY_BUF(0%), 32%)
310         IF (SORT_STATUS% AND 1%) = 0%                           &
               THEN PRINT "Error in INIT_SORT:"; SORT_STATUS%       &
\           GOTO 32767

320         FOR I% = 1% TO 10%
330            REC$ = STRING$(32%, 65% + (26. * RND))
340            KEY$ = REC$
350            PRINT KEY$
360            SORT_STATUS% = SOR$RELEASE_REC(KEY_AND_REC$)
370            IF (SORT_STATUS% AND 1%) = 0%                        &
               THEN PRINT "Error in RELEASE_REC:"; SORT_STATUS%     &
\           GOTO 32767
380         NEXT I%

400         SORT_STATUS% = SOR$SORT_MERGE
410         IF (SORT_STATUS% AND 1%) = 0%                           &
               THEN PRINT "Error in SORT_MERGE:"; SORT_STATUS%      &
\           GOTO 32767
420         PRINT "Starting release"
430         REC$ = "***********"
500         WHILE -1%
510            SORT_STATUS% = SOR$RETURN_REC(REC$,LCL%)
520            IF (SORT_STATUS% AND 1%) = 0%                        &
               THEN IF SORT_STATUS% <> SS$ENDOFFILE                 &
                THEN PRINT "Error in RETURN_REC:"; SORT_STATUS%     &
\                     GOTO 32767                                    &
                ELSE GOTO 32767
540            PRINT REC$
550         NEXT
600         SORT_STATUS% = SOR$END_SORT
610         IF (SORT_STATUS% AND 1%) = 0%                           &
               THEN PRINT "Error in END_SORT:"; SORT_STATUS%
32767       END
```

3.5 Sample VAX–11 BASIC Program (MERGE)

```
100         !
            ! This program demonstrates the BASIC calling sequences
            ! for the merge RECORD interface.
            !
            EXTERNAL INTEGER FUNCTION SOR$INIT_MERGE, SOR$RETURN_REC
\           EXTERNAL INTEGER CONSTANT SS$_ENDOFFILE
\           EXTERNAL INTEGER READ_REC, KOMPAR
\           MAP (RECBUF) STRING REC=80
\           MAP (OUTBUF) STRING REC2=80
\           COM (FILES) LONG ORDER, FILTAB(10)

300         ORDER = 4%
\           FILTAB(I%) = I% FOR I% = 1% TO ORDER

400         !
            ! First open all the input files.  Share the record buffer.
            !
            OPEN "A.DAT" FOR INPUT AS FILE #FILTAB(1%), SEQUENTIAL FIXED     &
               , ACCESS READ, MAP RECBUF
\           OPEN "B.DAT" FOR INPUT AS FILE #FILTAB(2%), SEQUENTIAL FIXED     &
               , ACCESS READ, MAP RECBUF
\           OPEN "C.DAT" FOR INPUT AS FILE #FILTAB(3%), SEQUENTIAL FIXED     &
               , ACCESS READ, MAP RECBUF
\           OPEN "D.DAT" FOR INPUT AS FILE #FILTAB(4%), SEQUENTIAL FIXED     &
               , ACCESS READ, MAP RECBUF

500         !
            ! Initialize the merge.  Pass the merge order, the largest
            !  record length, the compare routine address, and the
            !  input routine address.
            !
            SYS_STATUS% = SOR$INIT_MERGE(ORDER,0%,80%,,KOMPAR,READ_REC)
\           CALL SYS$EXIT(SYS_STATUS% BY VALUE)                              &
            IF (SYS_STATUS% AND 1%) = 0%

600         !
            ! Now loop getting merged records. SOR$RETURN_REC will
            !  call READ_REC when it needs input.
            !
            SYS_STATUS% = SOR$RETURN_REC (REC2,LENGTH%)
\           GOTO 700 IF SYS_STATUS% = SS$_ENDOFFILE
\           PRINT SEG$(REC2,1%,LENGTH%)
\           GOTO 600

700         !
            ! Now tell SORT that we are all done.
            !
            SYS_STATUS% = SOR$END_SORT
\           CALL SYS$EXIT(SYS_STATUS% BY VALUE)                              &
            IF (SYS_STATUS% AND 1%) = 0%

900         END
```

(continued on next page)

```
1000        FUNCTION INTEGER READ_REC(STRING RECX, INTEGER FILEX, SIZE)
            !
            ! This routine reads a record from one of the input files
            !  for merging.  It will be called by SOR$INIT_MERGE and by
            !  SOR$RETURN_REC.
            ! Parameters:
            !
            !    RECX.wl.ds   Buffer to hold the record after it
            !  is read in.
            !
            !    FILE.rl.r    Indicates which file the record is
            !  to be read from.  1 specifies the
            !  first file, 2 specifies the second
            !  etc.
            !
            !    LENGTH.wl.r  Is the actual number of bytes in
            !  the record.  This is set by READ_REC
            !

1010        ON ERROR GOTO 1900

1020        EXTERNAL INTEGER CONSTANT SS$_ENDOFFILE, SS$_NORMAL
\           MAP (RECBUF) STRING REC=80
\           COM (FILES) INTEGER ORDER, FILTAB(10)

1030        READ_REC = SS$_ENDOFFILE
\           IF (FILEX < 1%) OR (FILEX > ORDER) THEN FUNCTIONEXIT

1040        GET #FILTAB(FILEX)
\           RECX = REC
\           SIZE = RECOUNT
\           READ_REC = SS$_NORMAL
\           FUNCTIONEXIT

1900        ! Here if end of file.
            RESUME 1910
1910        READ_REC = SS$_ENDOFFILE

1990        FUNCTIONEND

2000        FUNCTION INTEGER KOMPAR(STRING REC1=20 BY REF, REC2=20 BY REF)
            !
            ! This routine compares two records.  It returns -1
            !  if the first record is smaller than the second,
            !  0 if the records are equal, and 1 if the first
            !  record is larger than the second.
            !

2010        KOMPAR = 0%
\           FUNCTIONEXIT IF REC1 = REC2

2020        KOMPAR = -1%
\           FUNCTIONEXIT IF REC1 < REC2

2030        KOMPAR = 1%
2090        FUNCTIONEND
```

3.6 Sample VAX–11 BLISS–32 Program (SORT)

```
MODULE TESTSUB (MAIN = FILEIO,
                LANGUAGE (BLISS32) ,
                IDENT = 'X01.01 '
                ) =
BEGIN
!
! ENVIRONMENT: STARLET OPERATING SYSTEM, USER MODE UTILITY
!--

FORWARD ROUTINE
    CREATE_OUTPUT,
    OPEN_FILE,
    OPEN_INPUT,
    CLOSE_FILE,
    GET_RECORD,
    PUT_RECORD,
    RAB_INIT,
    OUT_RAB_INIT;

LIBRARY 'SYS$LIBRARY:STARLET.L32';

BIND
    FILENAMEIN = UPLIT BYTE(%ASCII'R010SQ.DAT'),
    FILENAMEOUT = UPLIT BYTE(%ASCII'TEST.TMP');

LITERAL
    ERROR = 0,
    SUCCESS = 1,
    RECORD_SIZE = 80,      ! Max record size
    KEY_SIZE = 10;      ! Size of key field.

OWN
    COM_FAB : $FAB_DECL,    !FAB
    COM_RAB : $RAB_DECL,    !RAB
    OUT_FAB : $FAB_DECL,    !OUTPUT FAB
    OUT_RAB : $RAB_DECL,    !OUTPUT RAB
    FILEIN : VECTOR [2] INITIAL (10, FILENAMEIN),
    FILEOUT : VECTOR [2] INITIAL (8, FILENAMEOUT),
    OUTSIZ,
    KEYBUF : WORD INITIAL (1),
    KEYTYP : WORD INITIAL (1),
    KEYORD : WORD INITIAL (0),
    KEYPOS : WORD INITIAL (1),
    KEYSIZ : WORD INITIAL (KEY_SIZE),
    INLRL : WORD INITIAL (KEY_SIZE + RECORD_SIZE),
    WRKFILE : INITIAL (500),
    NUMWRK : BYTE INITIAL (4),
    BUFFER : VECTOR [RECORD_SIZE + KEY_SIZE, BYTE];

BIND
    KEYAREA = BUFFER [0] : VECTOR [KEY_SIZE, BYTE],
    RECORDBUF = BUFFER [KEYSIZE] : VECTOR [RECORD_SIZE, BYTE];
```

(continued on next page)

```
OWN
    RECDESC : VECTOR [2] INITIAL (KEY_SIZE + RECORD_SIZE, KEYAREA);


EXTERNAL ROUTINE
    SOR$PASS_FILES,
    SOR$INIT_SORT,
    SOR$SORT_MERGE,
    SOR$RELEASE_REC,
    SOR$RETURN_REC,
    SOR$END_SORT;

GLOBAL ROUTINE FILEIO : =
    BEGIN

    LOCAL
         ERRORCODE;

! TRY RECORD I/O

    IF NOT OPEN_INPUT () THEN RETURN ERROR;
    IF NOT RAB_INIT (0) THEN RETURN ERROR;
    OUT_RAB_INIT ();
    IF NOT SOR$INIT_SORT (KEYBUF, INLRL, WRKFILE, NUMWRK) THEN RETURN ERROR;

    WHILE 1 DO
            BEGIN
                                        ! Fill record with blanks.
            CH$FILL (' ', RECORD_SIZE, CH$PTR (RECORDBUF));
            ERRORCODE = GET_RECORD ();      ! Read a record.
            IF .ERRORCODE EQL RMS$_EOF THEN EXITLOOP;
            IF NOT .ERRORCODE THEN RETURN .ERRORCODE;
                                        ! Extract the key.
            CH$MOVE (KEY_SIZE, CH$PTR(RECORDBUF), CH$PTR(KEYAREA));
                                        ! Pass the record to SORT
            ERRORCODE = SOR$RELEASE_REC (RECDESC);
            IF NOT .ERRORCODE THEN RETURN .ERRORCODE;
            END;

    ERRORCODE = SOR$SORT_MERGE ();      ! Sort the records.
    IF NOT .ERRORCODE THEN RETURN .ERRORCODE;

    WHILE 1 DO
            BEGIN
                                        ! Retrieve a sorted record
            ERRORCODE = SOR$RETURN_REC (RECDESC, OUTSIZ);
            IF .ERRORCODE EQL SS$_ENDOFFILE THEN EXITLOOP;
            PUT_RECORD (.OUTSIZ, KEYAREA);      ! Copy to output file.
            END;

    ERRORCODE = SOR$END_SORT ();      ! Clean up.
    IF NOT .ERRORCODE THEN RETURN .ERRORCODE;

    CLOSE_FILE (1);
    SUCCESS
    END;
```

(continued on next page)

```
ROUTINE OPEN_INPUT : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE SETS UP THE FAB FOR THE INPUT
!                 FILE AND CALLS THE COMMON OPEN ROUTINE.
! FORMAL PARAMETERS:
!           NONE
! IMPLICIT INPUTS:
!           NONE
! IMPLICIT OUTPUTS:
!           NONE
! ROUTINE VALUE: AN ERROR CODE ON ERROR
! COMPLETION CODES: 1 FOR SUCCESS
! SIDE EFFECTS: THE INPUT FILE NAME IS PARSED AND THE FILE OPENED
!--
    BEGIN

    LITERAL
            DEFAULT_SIZE = 4, !DEFAULT TYPE SIZE
            KEY_BUCKET = 15,  !OVERHEAD IN INDEX BUCKET
            TOTAL_KEY = 3;    !OVERHEAD IN TOTAL KEY SIZE

    BIND
            DEFAULT_TYPE = UPLIT BYTE(%ASCII'.DAT');

! INITIALIZE THE FAB

    $FAB_INIT (FAB = COM_FAB, FNA = CH$PTR (FILENAMEIN),
       FNS = 10, DNA = CH$PTR (DEFAULT_TYPE),
    DNS = DEFAULT_SIZE, FOP = SQO, RAT = CR);
    IF NOT OPEN_FILE () THEN RETURN ERROR;
    CREATE_OUTPUT ()
    END;

ROUTINE RAB_INIT (LRL) : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE SETS UP THE RAB FOR THE INPUT
!                 FILE AND CONNECTS.
!
! FORMAL PARAMETERS: THE LONGEST RECORD LENGTH
! IMPLICIT INPUTS:
!           NONE
! IMPLICIT OUTPUTS:
!           NONE
! ROUTINE VALUE: THE TOTAL NEEDED BUFFER SIZE, OR AN ERROR CODE.
! COMPLETION CODES:
! SIDE EFFECTS:
!--
    BEGIN

! IF AN INDEX SORT CREATE THE OUTPUT FILE NOW THAT WE KNOW THE KEY SIZE.
! INITIALIZE THE RAB

    $RAB_INIT (RAB = COM_RAB, FAB = COM_FAB, RAC = SEQ, KRF = 0,
       UBF = RECORDBUF, USZ = RECORD_SIZE, ROP = RAH);
    IF NOT $RMS_CONNECT (RAB = COM_RAB) THEN RETURN ERROR;
    SUCCESS
    END;
```

(continued on next page)

```
ROUTINE CREATE_OUTPUT : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE SETS UP THE FAB FOR THE OUTPUT
!                 FILE AND CREATES IT.
! FORMAL PARAMETERS: NONE
! IMPLICIT INPUTS:
!           NONE
! IMPLICIT OUTPUTS:
!           NONE
! ROUTINE VALUE: AN ERROR CODE ON ERROR.
! COMPLETION CODES: 1 FOR SUCESS,
! SIDE EFFECTS: THE OUTPUT FILE NAME IS PARSED AND THE FILE CREATED.
!--
    BEGIN

! INITIALIZE THE FAB

    $FAB_INIT (FAB = OUT_FAB, FAC = PUT, FNA = CH$PTR (FILENAMEOUT),
        FNS = 8);

! DEFAULT TO INPUT FILE ORGANIZATION, RECORD FORMAT AND ATTRIBUTES.
!
    OUT_FAB [FAB$B_ORG] = .COM_FAB [FAB$B_ORG];
    OUT_FAB [FAB$B_RFM] = .COM_FAB [FAB$B_RFM];
    OUT_FAB [FAB$B_RAT] = .COM_FAB [FAB$B_RAT];
!
! FINALLY CREATE THE FILE .

    IF NOT $RMS_CREATE (FAB = OUT_FAB)  !CREATE OR OPEN FILE
    THEN
            RETURN ERROR;
    SUCCESS
    END;

ROUTINE OUT_RAB_INIT : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE SETS UP THE RAB FOR THE OUTPUT
!                 FILE AND CONNECTS.
!
! FORMAL PARAMETERS: NONE
! IMPLICIT INPUTS:
!           NONE
! IMPLICIT OUTPUTS:
!           NONE
! ROUTINE VALUE:
! COMPLETION CODES: ERROR CODE ON ERROR OR SUCCESS
! SIDE EFFECTS:
!--
    BEGIN

! INITIALIZE THE RAB BASED ON SORT TYPE AND FILE ORGANIZATION
!
    $RAB_INIT (RAB = OUT_RAB, KRF = 0, RAC = SEQ, ROP = WBH, FAB = OUT_FAB);
    IF NOT $RMS_CONNECT (RAB = OUT_RAB) THEN RETURN ERROR;
    SUCCESS
    END;
```

(continued on next page)

```
ROUTINE OPEN_FILE : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE CALLS THE RMS FILE NAME PARSER
!               AND OPENS THE FILE.
!
! FORMAL PARAMETERS: NONE
! IMPLICIT INPUTS:
!            NONE
! IMPLICIT OUTPUTS:
!            NONE
! ROUTINE VALUE:
! COMPLETION CODES: 1 FOR SUCCESS, 0 FOR ERROR
!--
    BEGIN
    IF NOT $RMS_OPEN (FAB = COM_FAB) THEN RETURN ERROR;
    SUCCESS
    END;

ROUTINE GET_RECORD : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE CALLS THE RMS GET RECORD ROUTINE
!               TO READ A RECORD FROM A FILE.
!
! FORMAL PARAMETERS: NONE
! IMPLICIT INPUTS: THE FAB AND RAB ALREADY INITIALIZED FOR THE
!                APPROPRIATE FILE.
! IMPLICIT OUTPUTS:
!            NONE
! ROUTINE VALUE: THE ADDRESS OF THE RECORD DESCRIPTOR, OR AN EOF STATUS.
! COMPLETION CODES: AN RMS ERROR
! SIDE EFFECTS: IF AN ERROR OCCURS A MESSAGE IS SENT TO THE USER.
!--
    BEGIN

! GET RECORDS UNTIL EITHER A RECORD IS READ OR END OF FILE IS REACHED.
! PRINT A WARNING MESSAGE FOR EVERY READ ERROR THAT OCCURS.

    WHILE 1 DO
            SELECTONE $RMS_GET (RAB = COM_RAB) OF
                SET
                [RMS$_EOF] :
                    BEGIN
                    CLOSE_FILE (0);
                    RETURN RMS$_EOF;
                    END;
                [RMS$_SUC] :
                    IF .COM_RAB [RAB$W_RSZ] NEQ 0
                    THEN
                        RETURN SUCCESS;
                [OTHERWISE] :
                    RETURN ERROR;
                TES;
    SUCCESS
    END;
```

(continued on next page)

```
ROUTINE PUT_RECORD (LEN, ADR) : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE CALLS THE RMS PUT RECORD ROUTINE
!               TO WRITE A RECORD TO A FILE.
! FORMAL PARAMETERS: THE LENGTH AND ADDRESS OF THE RECORD
! IMPLICIT INPUTS: THE FAB AND RAB ALREADY INITIALIZED FOR THE
!                 APPROPRIATE FILE.
! IMPLICIT OUTPUTS:
!           NONE
! ROUTINE VALUE:
! COMPLETION CODES: NONE
! SIDE EFFECTS: IF AN ERROR OCCURS A MESSAGE IS SENT TO THE USER.
!--
    BEGIN

! PUT THE RECORD INTO THE OUTPUT FILE.
!
    OUT_RAB [RAB$L_RBF] = .ADR;
    OUT_RAB [RAB$W_RSZ] = .LEN;
    IF NOT $RMS_PUT (RAB = OUT_RAB) THEN RETURN ERROR;
    SUCCESS
    END;

ROUTINE CLOSE_FILE (OUTPUT_FLAG) : =

!++
! FUNCTIONAL DESCRIPTION: THIS ROUTINE DISCONNECTS AND CLOSES A FILE.
!
! FORMAL PARAMETERS: A FLAG INDICATING WHETHER TO CLOSE THE OUTPUT FILE,
!             AS WELL AS THE INPUT FILE OR DELETE IT.
! IMPLICIT INPUTS: THE RAB AND FAB INITIALIZED FOR THE APPROPRIATE
!                 FILE.
! IMPLICIT OUTPUTS:
!           NONE
! ROUTINE VALUE:
! COMPLETION CODES: NONE
! SIDE EFFECTS: IF AN ERROR OCCURS A MESSAGE IS SENT TO THE USER.
!--
    BEGIN
    LITERAL
            CLOSE_IN = 0,
            CLOSE_BOTH = 1,
            DELETE_OUT = 2;

! BASED ON THE OUTPUT FLAG EITHER CLOSE THE OUTPUT FILE AND THE INPUT
! FILE, CLOSE THE INPUT FILE OR CLOSE THE INPUT FILE AND DELETE THE
! OUTPUT FILE. RETURN ANY ERRORS.

    IF NOT .OUTPUT_FLAG
    THEN
            BEGIN
            IF NOT $RMS_DISCONNECT (RAB = COM_RAB) THEN RETURN ERROR;
            IF NOT $RMS_CLOSE (FAB = COM_FAB) THEN RETURN ERROR;
            END;
    IF .OUTPUT_FLAG
    THEN
            BEGIN
            IF NOT $RMS_DISCONNECT (RAB = OUT_RAB) THEN RETURN ERROR;
            IF NOT $RMS_CLOSE (FAB = OUT_FAB) THEN RETURN ERROR;
            END
```

(continued on next page)

```
    ELSE
            IF .OUTPUT_FLAG EQL DELETE_OUT
            THEN
                IF NOT $RMS_ERASE (FAB = OUT_FAB) THEN RETURN ERROR;
    SUCCESS
    END;
END

ELUDOM
```

3.7 Sample VAX–11 FORTRAN Program (SORT)

```
      PROGRAM CALLSORT
C
C
C     THIS IS A SAMPLE FORTRAN PROGRAM THAT CALLS THE
C     NATIVE SORT SUBROUTINE PACKAGE USING THE FILE I/O INTERFACE.
C     THIS PROGRAM REQUESTS AN INDEX SORT OF FILE 'R010SQ.DAT'
C     INTO THE FILE 'TEST.TMP', THE KEY IS AN 80 BYTE CHARACTER
C     ASCENDING KEY STARTING IN POSITION ONE OF EACH RECORD.
C
C
C     DEFINE EXTERNAL FUNCTIONS AND DATA
C
      CHARACTER*10 INPUTNAME !INPUT FILE NAME
      CHARACTER*8 OUTPUTNAME !OUTPUT FILE NAME
      INTEGER*2 KEYBUF(5) !KEY DEFINITION BUFFER
      INTEGER*2 NUMWRK !NUMBER OF WORK FILES
      INTEGER*2 ISRTTYP !SORT PROCESS
      INTEGER*4 SOR$PASS_FILES !SORT FUNCTION NAMES
      INTEGER*4 SOR$INIT_SORT
      INTEGER*4 SOR$SORT_MERGE
      INTEGER*4 SOR$END_SORT
      INTEGER*4 ISTATUS !STORAGE FOR SORT FUNCTION VALUE
C
C     INITIALIZE DATA - FIRST THE FILENAMES THEN THE KEY BUFFER FOR
C     ONE 80 BYTE CHARACTER KEY STARTING POSITION 1, 3 WORK FILES
C     AND AN INDEX SORT PROCESS
C
      DATA INPUTNAME,OUTPUTNAME/'R010SQ.DAT','TEST.TMP'/
      DATA KEYBUF,NUMWRK,ISRTTYP/1,1,0,1,80,3,3/
C
C     CALL THE SORT EACH CALL IS A FUNCTION
C
C
C     PASS SORT THE FILENAMES
C
      ISTATUS = SOR$PASS_FILES(INPUTNAME,OUTPUTNAME)
      IF (.NOT. ISTATUS) GOTO 10
C
C     INITIALIZE WORK AREAS AND KEYS
C
      ISTATUS = SOR$INIT_SORT(KEYBUF,,,NUMWRK,ISRTTYP)
      IF (.NOT. ISTATUS) GOTO 10
C
C     SORT THE RECORDS
C
      ISTATUS = SOR$SORT_MERGE( )
      IF (.NOT. ISTATUS) GOTO 10
C
C     CLEAN UP WORK AREAS AND FILES
C
      ISTATUS = SOR$END_SORT()
      IF (.NOT. ISTATUS) GOTO 10
      STOP 'SORT SUCCESSFUL'
10    STOP 'SORT UNSUCCESSFUL'
      END
```

3.8 Sample VAX–11 FORTRAN Program (MERGE)

```
C...
C...    This program demonstrates the FORTRAN calling sequences
C...    for the merge RECORD interfaces.
C...

        IMPLICIT INTEGER (A-Z)

        PARAMETER ORDER=4    ! Order of the merge.
        PARAMETER EOF=2160    ! End of file status code.
        EXTERNAL READ_REC   ! Routine to read a record.
        EXTERNAL KOMPAR     ! Routine to compare records.
        CHARACTER*80 REC    ! A record.


C...
C...    First open all the input files.
C...
        OPEN (UNIT=10, FILE='DBB1:X.DAT',TYPE='OLD',READONLY,
      * FORM='FORMATTED')
        OPEN (UNIT=11, FILE='DBB1:Y.DAT',TYPE='OLD',READONLY,
      * FORM='FORMATTED')
        OPEN (UNIT=12, FILE='DBB1:Z.DAT',TYPE='OLD',READONLY)
        OPEN (UNIT=13, FILE='DBB1:A.DAT',TYPE='OLD',READONLY)
        OPEN (UNIT=8, FILE='SYS$OUTPUT:', TYPE='NEW') ! Open the output file.

C...
C...    Initialize the merge.  Pass the merge order, the largest
C...    record length, the compare routine address, and the
C...    input routine address.
C...
        ISTAT = SOR$INIT_MERGE (ORDER, 0, 80,,KOMPAR,READ_REC)
        IF (.NOT. ISTAT) GOTO 10   ! Check for error.


C...
C...    Now loop getting merged records.  SOR$RETURN_REC will
C...    call READ_REC when it needs input.
C...
5       ISTAT = SOR$RETURN_REC (REC, LENGTH)
        IF (ISTAT .EQ. EOF) GO TO 30   ! Check for end of file.
        IF (.NOT. ISTAT) GO TO 10   ! Check for error.

        WRITE(8,200) REC    ! Output the record.
200     FORMAT(' ',A)
        GOTO 5      ! And loop back.
C...
C...    Now tell SORT that we are all done.
C...

30      ISTAT = SOR$END_SORT()
        IF (.NOT. ISTAT) GOTO 10   ! Check for error.
        CALL EXIT

C...
C...    Here if an error occurred.  Write out the error status
C...    and exit.
C...
```

(continued on next page)

```
10      WRITE(8,201)ISTAT
201     FORMAT(' ?ERROR CODE', I20)
        CALL EXIT
        END
        FUNCTION READ_REC (RECX, FILE, SIZE)
C...
C...    This routine reads a record from one of the input files
C...    for merging. It will be called by SOR$INIT_MERGE and by
C...    SOR$RETURN_REC.
C...     Parameters:
C...
C...         RECX.wcp.ds    character buffer to hold the record after
C...                        it is read in.
C...
C...         FILE.rl.r      indicates which file the record is
C...                        to be read from.  1 specifies the
C...                        first file, 2 specifies the second
C...                        etc.

C...
C...         LENGTH.wl.r    is the actual number of bytes in
C...                        the record.  This is set by READ_REC.
C...
        IMPLICIT INTEGER (A-Z)
        PARAMETER EOF=2160    ! End of file status code.
        PARAMETER SUCCESS=1    ! Success status code.
        PARAMETER MAXFIL=10    ! Max number of files.


        LOGICAL*1 FILTAB(MAXFIL)
        CHARACTER*(*) RECX    ! Passed length character argument.

        DATA FILTAB/10,11,12,13,14,15,16,17,18,19/ ! Table of I/O unit
             numbers.

        READ_REC = EOF     ! Give end of file return
        IF (FILE .LT. 1 .OR. FILE .GT. MAXFIL) RETURN  ! If illegal call.

        READ (FILTAB(FILE), 100, END=50) RECX ! Read the record.
100     FORMAT(A)


        READ_REC = SUCCESS    ! Return success code.
        SIZE = LEN (RECX)    ! Return size of record.
        RETURN


C...    Here if end of file.
50      READ_REC = EOF     ! Return "end of file" code.
        RETURN
        END
        FUNCTION KOMPAR (REC1,REC2)

C...
C...    This routine compares two records.  It returns -1
C...    if the first record is smaller than the second,
C...    0 if the records are equal, and 1 if the first record
C...    is larger than the second.
C...
```

(continued on next page)

```
      PARAMETER KSIZ=20

      IMPLICIT INTEGER (A-Z)

      LOGICAL*1 REC1(KSIZ),REC2(KSIZ)

      DO 20 I=1,KSIZ

      KOMPAR = REC1(I) - REC2(I)
      IF (KOMPAR .NE. 0) GOTO 50

20    CONTINUE

      RETURN


50    KOMPAR = ISIGN (1, KOMPAR)
      END
```

3.9 Sample VAX–11 MACRO Program (SORT)

```
    .TITLE TESTSUB
    .IDENT x01.01
;
; THIS IS A SAMPLE MACRO
; PROGRAM WHICH CALLS
; THE SORT SUBROUTINE
; PACKAGE. THERE IS AN
; EXAMPLE USING EACH
; INTERFACE.
;
;
; DATA AREA
;
FILENAMEIN:     .ASCII   /R010SQ.DAT/      ;INPUT FILENAME
FILENAMEOUT:    .ASCII   /TEST.TMP/        ;OUTPUT FILENAME
                .BLKB    2
IN_FAB:         .BLKB    80                ;RMS DATA BLOCKS
IN_RAB:         .BLKB    68
OUT_FAB:        .BLKB    80
OUT_RAB:        .BLKB    68
FILEIN:         .LONG    10                ;INPUT FILE NAME DESCRIPTOR
                .ADDRESS FILENAMEIN
FILEOUT:        .LONG    8                 ;OUTPUT FILE NAME DESCRIPTOR
                .ADDRESS FILENAMEOUT
KEYBUF:         .WORD    1                 ;KEY DEFINITION BUFFER
KEYTYPE:        .WORD    1
KEYORD:         .WORD    0
KEYPOS:         .WORD    1
KEYSIZ:         .WORD    10
INLRL:          .WORD    80                ;INPUT RECORD LONGEST LENGTH
WRKFILE:        .LONG    500               ;WORK FILE SIZE
NUMWRK:         .BYTE    4                 ;NUMBER OF WORK FILES
TAGSRT:         .BYTE    2                 ;TAG SORT
                .BKLB    2
KEYAREA:        .BKLB    10                ;KEY BUFFER
RECORDBUF:      .BKLB    80                ;RECORD BUFFER
RECDESC:        .LONG    90                ;RECORD DESCRIPTOR
    .ADDRESS KEYAREA

;
;
;
; FIRST THE FILE I/O INTERFACE. DO A TAG SORT ON THE FILE 'R010SQ.DAT'
; INTO THE FILE 'TEST.TMP' USING 4 WORK FILES. KEY IS CHARACTER, 10 BYTES
; LONG, STARTING POSITION 1.
;
;
    .EXTRN      SOR$PASS_FILES,SOR$INIT_SORT,SOR$SORT_MERGE,SOR$END_SORT,-
                SOR$RELEASE_REC,SOR$RETURN_REC
;
FILEIO::
    .ENTRY      ^M<R2,R3,R4,R5,R6,R7>  ;SAVE REGISTERS
                                       ;DEFAULT ALL OUTPUT OPTIONS
    PUSHAB      FILEOUT                ;PUSH FILENAME DESCRIPTOR ADDRESS
    PUSHAB      FILEIN
    CALLS       #2,SOR$PASS_FILES      ;PASS FILENAMES TO SORT
    BLBC        R0,2$                  ;TEST FOR ERROR
    PUSHAB      TAGSRT                 ;PUSH SORT TYPE
    PUSHAB      NUMWRK                 ;PUSH NUMBER OF WORK FILES
    CLRQ        (SP)                   ;DEFAULT LRL AND WORK FILE SIZE
    PUSHAB      KEYBUF                 ;PUSH KEY BUFFER ADDRESS
    CALLS       #5,SOR$INIT_SORT       ;INITIALIZE THE SORT
    BLBC        R0,2$                  ;TEST FOR ERROR
                                       LET SORT DO COMPARES
```

(continued on next page)

```
    CALLS       #0,SOR$SORT_MERGE       ;START SORTING
    BLBC        R0,2$                   ;TEST FOR ERROR
    CALLS       #0,SOR$END_SORT         ;DO CLEAN UP
    BLBC        R0,2$                   ;TEST FOR ERROR
;
;
; NOW TRY THE RECORD I/O INTERFACE. RECORDS ARE 80 BYTES LONG, KEY IS
; CHARACTER, 10 BYTES LONG, STARTING IN POSITION 1. WORK FILE SIZE IS
; 500 BLOCKS.
;
;
    CALLS       #0,OPEN_INPUT           ;OPEN USER INPUT AND OUTPUT FILE
    BLBC        R0,2$                   ;TEST FOR ERROR
                                        ;DEFAULT SORT TYPE AND WORK FILES
    PUSHAB      WRKFILE                 ;PUSH WORK FILE SIZE
    PUSHAB      INLRL                   ;PUSH LRL
    PUSHAB      KEYBUF                  ;PUSH KEY BUFFER ADDRESS
    CALLS       #3,SOR$INIT_SORT        ;INITIALIZE THE SORT
    BLBC        R0,2$                   ;TEST FOR ERROR
    MOVZWL      #1000,R6                ;SET UP LOOP INDEX
1$: CALLS       #0,GET_RECORD           ;GET RECORD FROM MY FILE
    BLBC        R0,2$                   ;TEST FOR ERROR
    MOVC3       #10,RECORDBUF,KEYAREA   ;SET UP KEY IN KEY BUFFER
                                        ;SORT DOES COMPARES
    PUSHAB      RECDESC                 ;PUSH RECORD DESCRIPTOR
    CALLS       #1,SOR$RELEASE_REC      ;GIVE RECORD TO SORT
    BLBC        R0,2$                   ;TEST FOR ERROR
    SOBGTR      R6,1$
                                        ;SORT DOES COMPARES
    CALLS       #0,SOR$SORT_MERGE       ;NO MORE RECORDS TO GIVE
2$: BLBC        R0,6$
3$:
    PUSHAB      INLRL                   ;PUSH RECORD SIZE LOCATION
    PUSHAB      RECDESC                 ;PUSH RECORD DESCRIPTOR
    CALLS       #2,SOR$RETURN_REC       ;GET RECORD BACK
    CMPL        R0,SS#_ENDOFFILE        ;GOTTEN ALL RECORDS
    BEQL        4$                      ;YES
    BLBC        R0,6$                   ;ERROR
    CALLS       #0,PUT_RECORD           ;PUT RECORD INTO OUTPUT
    BRB         3$
4$: CALLS       #0,SOR$END_SORT         ;FINISH UP
    BLBC        R0,6$                   ;TEST FOR ERROR
    CALLS       #0,CLOSE_FILE           ;CLOSE UP FILES
    MOVL        #1,R0                   ;INDICATE SUCCESS
    RET
6$: CLRL        R0                      ;INDICATE FAILURE
    RET
;
    .END
```

3.10 Sample VAX–11 PASCAL Program (SORT)

```
(* Example of a program using SORT. Taken from a FORTRAN example in the SORT
USER'S GUIDE.                                                              *)

program sort(output) ;

label 13,   (* error exit *)
      999 ; (* program end *)

const

(**** Common sort definitions ****)

   ss$normal=          1 ;

(* Returns from sor$init_sort.                                             *)

   sor$_sort_on=       %x1c802c ;
   sor$_miss_key=      %x1c8004 ;
   sor$_bad_type=      %x1c806c ;
   sor$_bad_lrl=       %x1c8084 ;
   sor$_lrl_miss=      %x1c8074 ;
   sor$_bad_file=      %x1c808c ;
   sor$_work_dev=      %x1c800c ;
   sor$_vm_fail=       %x1c801c ;
   sor$_ws_fail=       %x1c8024 ;
   sor$_num_key=       %x1c803c ;
   sor$_key_len=       %x1c80ac ;

(* Returns from sor$pass_files.                                            *)

   sor$_var_fix=       %x1c8064 ;
   sor$_inconsis=      %x1c805c ;
   sor$_openin=        %x1c109c ;
   sor$_openout=       %x1c10a4 ;

(* Returns from sor$release_rec.                                           *)

   sor$_bad_adr=       %x1c8094 ;
   sor$_extend=        %x1c80a4 ;
   sor$_map=           %x1c809c ;
   sor$_no_wrk=        %x1c8014 ;

(* Returns from sor$sort_merge.                                            *)

   sor$_readerr=       %x1c10b4 ;
   sor$_writeerr=      %x1c10d4 ;
   sor$_badfield=      %x1c101c ;

(* Returns from sor$return_rec                                             *)

   ss$_endoffile=      %x870 ;

(* Returns from sor$end_sort                                               *)

   sor$_clean_up=      %x1c80b4 ;
```

(continued on next page)

```
(* Key types                                                          *)

   character=          1 ;
   binary=             2 ;
   zoned=              3 ;
   packed_decimal=     4 ;
   dec_lead_overpun=   6 ;
   dec_lead_sep=       7 ;
   dec_trail_overpun=  8 ;
   dec_trail_sep=      9 ;
   float=             10 ;
   double_float=      11 ;
   g_float=           12 ;
   h_float=           13 ;

(* Key orders                                                         *)

   ascending=          0 ;
   descending=         1 ;

(* Sort types                                                         *)

   recordsort=         1 ;
   tagsort=            2 ;
   indexsort=          3 ;
   addresssort=        4 ;

(**** End of general sort definitions ****)

   numberofKeys=       1 ;    (* Number of Keys for this sort        *)

type

   byte=           0..255 ;
   word=           0..65535 ;

   Keybufferblock= packed record
         Keytype:    word ;
        Keyorder:    word ;
   startposition:    word ;
          length:    word
      end ;


(* The Keybuffer. Note that the field 'buffer' is a one element array in
   this program. This shows how a multi-Keyed sort would be defined.  *)

   Keybuffer= packed record
                   numKeys:  word ;
                   blocks:  packed array[1..numberofKeys] of Keybufferblock
              end ;

(* Name types for input and output files. A necessary fudge for %stdescr
   mechanism.                                                         *)

   inputnametype= packed array[1..10] of char ;
  outputnametype= packed array[1..8] of char ;

var

       buffer:  Keybuffer ;     (* the actual Keybuffer               *)
       status:  integer ;       (* return status                      *)
     sorttype:  byte ;          (* sort direction (ascending or descending)*)
 numworkfiles:  byte ;          (* number of work files               *)
    inputname:  inputnametype ;                 (* input file name    *)
   outputname:  outputnametype ;                (* output file name   *)
```

(continued on next page)

```
function sor$pass_files(  %stdescr inname:  inputnametype ;
                          %stdescr outname:  outputnametype ) : integer ;
                                                               extern ;

function sor$init_sort(      var buffer:  keybuffer ;
                             %immed lrl:  word ;
                        %immed filesize:  integer ;
                        var numworkfiles:  byte ;
                            var sorttype:  byte ) : integer ; extern ;

function sor$sort_merge : integer ; extern ;

function sor$end_sort : integer ; extern ;


begin

   (* initialize data for one 80 byte character key, starting position 1,
      3 work files, and a record sort process                              *)

   inputname := 'R010SQ.DAT' ;
   outputname := 'TEST.TMP' ;
   with buffer do begin
   numkeys := 1 ;
      with blocks[1] do begin
         keytype := character ;
         keyorder := ascending ;
         startposition := 1 ;
         length := 80
      end
   end ;
   numworkfiles := 3 ;
   sorttype := recordsort ;

   (* call the sort as a series of functions                               *)

   (* pass filenames to sort                                               *)

   status := sor$pass_files( inputname, outputname ) ;
   if not odd(status) then begin
      (* process error, Here we just fail *)
      goto 13
   end ;

   (* initialize work areas and keys                                       *)

   status := sor$init_sort( buffer, 0, 0, numworkfiles, sorttype ) ;
   if not odd(status) then begin
      (* process error, Here we just fail *)
      goto 13
   end ;

   (* sort the records                                                     *)

   status := sor$sort_merge ;
   if not odd(status) then begin
      (* process error, Here we just fail *)
      goto 13
   end ;

   (* clean up work areas and files                                        *)
```

(continued on next page)

```
    status := sor$end_sort ;
    if not odd(status) then begin
      (* process error. Here we just fail *)
      goto 13
    end ;

    writeln( 'Sort successful.' ) ;
    goto 999 ;

  13: writeln( 'Sort unsuccessful.' ) ;
  999: ;
end.
```

3.11 Sample VAX–11 PASCAL Program (MERGE)

```
(* This program merges 3 input files into one output file using the
   record interface, and a user-defined key comparison routine. *)
(*$o-*)
program merge( output, infile1, infile2, infile3, outfile ) ;

const
   merge_order = 3 ;  (* Number of input files. Must match number of
                         infiles declared in 'program' statement
                         and number of input files opened in procedure
                         'initfiles', and the case statement in
                         function 'readrecord'. *)
   longest_record_length = 131 ;

   ss$_normal = 1 ;
   ss$_endoffile = %x870 ;

type
   byte = 0..255 ;
   word = 0..65535 ;

   record_buffer = packed array[1..longest_record_length] of char ;
   record_buffer_descr = packed record
                           length : integer ;
                           address : Record_buffer ;
                         end ;
var
   infile1,     (* input files *)
   infile2,
   infile3,
   outfile : text ;  (* output file *)
   rec : record_buffer_descr ;    (* A record descriptor *)
   record_length : word ;         (* Length returned from sor$return_rec *)
   status,                        (* Function return status code *)
   i : integer ;                  (* Loop control variable *)


function Keycompare( rec1, rec2 : record_buffer ) : integer ;
(* Compare key field of rec1 to rec2. Merge is in ascending order on the
   first 'keylen' ascii characters of each record. This routine
   assumes that each record is at least 'keylen' characters long. *)

const
   Keylen = 5 ;
var
   Key1, Key2 : packed array[1..keylen] of char ;

begin (* Keycompare *)
   for i := 1 to Keylen do
      begin
      Key1[i] := rec1[i] ;
      Key2[i] := rec2[i] ;
      end ;
   if Key1 < Key2 then
      Keycompare := -1
   else if Key1 = Key2 then
      Keycompare := 0
   else
      Keycompare := 1 ;
end ; (* Keycompare *)
```

(continued on next page)

```
function readrecord(     var rec : record_buffer_descr ;
                       filenumber : integer ;
                    var recordsize : word ) : integer ;

(* Read one record from the specified file *)

procedure readone( var filename : text ) ;

(* read one record from filename. *)
var
   ch : char ;

begin (* readone *)
   recordsize := 0 ;  (* initialize record size *)
   if eoln (filename) and not eof (filename) then
            read (filename, ch) ;

   if eof( filename ) then
      readrecord := ss$_endoffile      (* return eof *)
   else                                (* read one record *)

    while not eoln( filename ) do
            if recordsize < rec.length then
            begin
            read( filename, ch ) ;
                recordsize := recordsize + 1 ;
            rec.address[recordsize] := ch ;
            end ;


end ; (* readone *)

begin (* readrecord *)

   readrecord := ss$_normal ; (* assume no end of file *)
   case filenumber of          (* read from specified file *)
      1: readone( infile1 ) ;
      2: readone( infile2 ) ;
      3: readone( infile3 )
      otherwise
         readrecord := ss$_endoffile  (* invalid file number *)
   end ;

end ; (* readrecord *)


procedure initfiles ;

begin
   (* open input files: sequential access, variable length records,
      RMS CR carriage control *)
   open( infile1, 'INFILE1.DAT', old ) ;
   open( infile2, 'INFILE2.DAT', old ) ;
   open( infile3, 'INFILE3.DAT', old ) ;

   (* open output file: same attributes as input files, except that
      it is a new file *)
   open( outfile, 'OUTFILE.DAT' ) ;

   (* ready input files for reading *)
   reset( infile1 ) ;
   reset( infile2 ) ;
   reset( infile3 ) ;
```

(continued on next page)

```
    (* ready output file for writing *)
    rewrite( outfile ) ;

end ; (* initfiles *)


(* Merge function declarations. All parameters
   are passed using by-reference semantics. *)


function sor$init_merge( merge_order : byte ;     (* number of files
                                                      to merge *)
                    %immed keybuffer  : integer ; (* not used *)
               longest_record_length : word ;
                %immed merge_options : integer ;   (* not used *)
          %immed function keycompare : integer ;
          %immed function readrecord : integer
                       ) : integer ; extern ;


function sor$return_rec(      var rec : record_buffer_descr ;
                         var record(size : word ) : integer ; extern ;


function sor$end_sort : integer ; extern ;


procedure error( status : integer ) ;
(* write error message and exit thru sys$exit *)

    procedure sys$exit( status : integer ) ; extern ;

begin (* error *)
    writeln( 'Merge unsuccessful. Status= hex ', status:8 hex ) ;
    sys$exit( status ) ;
end ; (* error *)


begin (* merge *)

    initfiles ; (* open and ready all files *)

    (* Initialize the merge. Pass the merge order, the key comparison
       routine address, the input routine address, and the longest
       record length *)
    status := sor$init_merge( merge_order, 0, longest_record_length,
                              0, keycompare, readrecord ) ;
    if not odd( status ) then
       error( status ) ;


    (* Loop getting merged records. SOR$RETURN_REC will call READRECORD
       when it needs input *)

    new( rec.address ) ;     (* get a record buffer *)
    rec.length := longest_record_length ; (* save buffer length *)

    repeat
       status := sor$return_rec( rec, record_length ) ;
       if odd( status ) then (* Success, write record to output file *)
          begin
          for i := 1 to record_length do
             write( outfile, rec.address[i] ) ;
          writeln (outfile) ;
          end ;
```

(continued on next page)

```
    until not odd( status ) ;
    if status <> ss$_endoffile then
       error( status ) ;

    status := sor$end_sort ;
    if not odd( status ) then
        error( status ) ;

    writeln( 'Merge successful.' ) ;

end. (* merge *)
```

3.12 Sample VAX–11 PL/I Program (SORT)

```
/*
  This program calls the SORT routines to perform an alphabetic
  sort on a file.
*/
SORTEM: PROCEDURE RETURNS (FIXED);

/*
  Include the declarations of the SORT procedures required
  for a sort using the file interface.
*/
%INCLUDE SOR$PASS_FILES;
%INCLUDE SOR$INIT_SORT;
%INCLUDE SOR$SORT_MERGE;
%INCLUDE SOR$END_SORT;
%INCLUDE $STSDEF;

/*
  Declare the input and output files; these are logical names
  that must be defined before the program is run.
*/
DECLARE INPUT_FILE CHARACTER(6) STATIC INIT('INFILE'),
        OUTPUT_FILE CHARACTER(7) STATIC INIT('OUTFILE');

/*
  Declare the key buffer array required to sort the first 80
  characters of any record. This 'array' is declared as a
  structure to clarify the example. An array can also be used.
*/
DECLARE 1 KEY_BUFFER STATIC,
          2 NUMBER_OF_KEYS FIXED BINARY (15) INIT(1),
          2 KEY_TYPE FIXED BINARY (15) INIT(1), /* character */
          2 KEY_ORDER FIXED BINARY (15) INIT(0), /* ascending order */
          2 START_POS FIXED BINARY(15) INIT(1),
          2 KEY_LENGTH FIXED BINARY(15) INIT(80),
        LONGEST_RECORD FIXED BINARY(15) STATIC INIT(80);

/*
  Declare global symbol names for RMS values to define the output file
*/
DECLARE (FAB$C_VAR, FAB$C_REL) GLOBALREF FIXED BINARY(31) VALUE;

DECLARE RECORD_TYPE FIXED BIN(7),
        FILE_ORG FIXED BINARY(7);

        RECORD_TYPE = FAB$C_VAR;
        FILE_ORG = FAB$C_REL;

/*
  Call the SORT routines in the required order.
  After each call to SORT, check STS$SUCCESS.
*/
        STS$VALUE = SOR$PASS_FILES(INPUT_FILE,OUTPUT_FILE,
                                   FILE_ORG,RECORD_TYPE);
        IF ^STS$SUCCESS THEN GOTO ERROR;

        STS$VALUE = SOR$INIT_SORT(KEY_BUFFER,LONGEST_RECORD);
        IF ^STS$SUCCESS THEN GOTO ERROR;

        STS$VALUE = SOR$SORT_MERGE();
        IF ^STS$SUCCESS THEN GOTO ERROR;

        STS$VALUE = SOR$END_SORT();
        IF ^STS$SUCCESS THEN GOTO ERROR;
```

(continued on next page)

```
        RETURN(1);          /* successful completion */

/*
  All errors return here with the value of STS$VALUE.
*/
ERROR:
        PUT SKIP(2) EDIT ('SORT failed. Error code',STS$VALUE)
                         (A,X,F(8));
        RETURN(STS$VALUE);
END;
```

3.13 Sample VAX–11 PL/I Program (SORT)

```
/*
  This program sorts the file STATE_FILE based on the field
  CAPITAL.NAME in each record.

  Logical name equivalences are required for the input file STATE_FILE
  and an output file SORTED_FILE.
*/
statesort: procedure options (main) returns (fixed binary(31));

/*
  Declare SORT routines
*/
%INCLUDE SOR$INIT_SORT;

%INCLUDE SOR$SORT_MERGE;

%INCLUDE SOR$END_SORT;

%INCLUDE SOR$RELEASE_REC;

%INCLUDE SOR$RETURN_REC;

%INCLUDE $STSDEF;

DECLARE SS$_ENDOFFILE GLOBALREF FIXED BINARY(31) VALUE,
        EOF BIT(1);
        EOF = '0'B;
/*
  Key buffer and data for SORT routines
*/
DECLARE 1 KEY_BUFFER STATIC,
          2 NUMBER_OF_KEYS FIXED BINARY (15) INIT(1),
          2 KEY_TYPE FIXED BINARY (15) INIT(1), /* character keys */
          2 KEY_ORDER FIXED BINARY (15) INIT(0), /* ascending order */
          2 START_POS FIXED BINARY(15) INIT(25),
          2 KEY_LENGTH FIXED BINARY(15) INIT(20),
        LONGEST_RECORD FIXED BINARY(15) STATIC INIT(196),
        KEYFIELD FIXED BINARY(7),        /* Code to decide sort */
        RETURN_LENGTH FIXED BINARY(15); /* Required parameter */
/* Declare a buffer to construct each record to be passed to SORT */

DECLARE 1 STATE_RECORD,          /* complete record */
          2 KEYFIELD CHARACTER(20),
          2 STATE,
            3 NAME CHARACTER (20),
            3 POPULATION FIXED BINARY(31),
            3 CAPITAL,
              4 NAME CHARACTER (20),
              4 POPULATION FIXED BINARY(31),
            3 LARGEST_CITIES(2),
              4 NAME CHARACTER(30),
              4 POPULATION FIXED BINARY(31),
            3 SYMBOLS,
              4 FLOWER CHARACTER (30),
              4 BIRD CHARACTER (30);

DECLARE 1 RECORD_DESCRIPTOR,
          2 LENGTH FIXED BINARY(15),
          2 FILLER FIXED BINARY(15),
          2 ADDRESS POINTER;
```

(continued on next page)

```
/*
  Input and output files
*/
DECLARE STATE_FILE FILE INPUT RECORD SEQUENTIAL,
        SORTED_FILE FILE RECORD OUTPUT SEQUENTIAL;
/*
  call SOR$INIT_SORT
*/
        STS$VALUE = SOR$INIT_SORT(KEY_BUFFER,LONGEST_RECORD);
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);
/*
   Enter DO-loop to read the input file STATE_FILE. Then, call
   SOR$RELEASE_REC. The record consists of the key field concatenated
   with the contents of STATE.
*/
        OPEN FILE(STATE_FILE);
        RECORD_DESCRIPTOR.LENGTH = 196;
        RECORD_DESCRIPTOR.ADDRESS = ADDR(STATE_RECORD);
        ON ENDFILE(STATE_FILE) EOF = '1'B;
        READ FILE (STATE_FILE) INTO(STATE);
        DO WHILE (^EOF);
           STATE_RECORD.KEYFIELD = CAPITAL.NAME;
            STS$VALUE = SOR$RELEASE_REC(
                RECORD_DESCRIPTOR) ;
            IF ^STS$SUCCESS THEN RETURN(STS$VALUE);
            READ FILE(STATE_FILE) INTO(STATE);
            END;
        CLOSE FILE(STATE_FILE);
PUT SKIP LIST('**** ALL RECORDS RELEASED');
/*
  Call SOR$SORT_MERGE to sort the records that were released
*/
        STS$VALUE = SOR$SORT_MERGE();
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);
/*
  Loop through the DO-group to get back each record and write it
  to the sorted output file.
*/
        STS$VALUE = 1;
        OPEN FILE(SORTED_FILE) OUTPUT;
        RECORD_DESCRIPTOR.LENGTH = 176;
        RECORD_DESCRIPTOR.ADDRESS = ADDR(STATE);
        DO WHILE (STS$VALUE ^= SS$_ENDOFFILE);
            STS$VALUE = SOR$RETURN_REC(RECORD_DESCRIPTOR,RETURN_LENGTH);
            IF STS$SUCCESS THEN WRITE FILE(SORTED_FILE) FROM (STATE);
            ELSE IF ^STS$SUCCESS & (STS$VALUE ^= SS$_ENDOFFILE)
                THEN RETURN(STS$VALUE);
            END;
        CLOSE FILE (SORTED_FILE);
/*
  Call SOR$END_SORT to finish up
*/
        STS$VALUE = SOR$END_SORT();
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE) ;
        RETURN(1);          /* successful completion */
/*
  All errors come here to return to the command level with the
  status value of the error in R0.
*/
ERROR: RETURN(STS$VALUE);
END;
```

Chapter 4
Improving SORT Efficiency

This chapter provides a brief description of how SORT operates internally, explains the meaning and use of SORT statistics, and suggests procedures the user and system manager can follow to make a sorting operation more efficient.

4.1 How SORT Operates

The following description of SORT's internal operation does not try to give all the details of the process. Instead, it gives a summary of what happens during a sorting operation, paying special attention to the sort tree and work files because these can be modified by the user for greater SORT efficiency.

SORT takes two forms: (1) a control program called the utility that the user invokes with a command line and (2) a package of subroutines the user can call from a program. During a utility sort the control program calls the subroutines as part of its processing. Although there are some differences in operation between the two forms, both go through the main phases described below.

4.1.1 Initialization Phase

SORT first reads and interprets the command line, specification file, or SOR$INIT_SORT parameters and stores the parameters and qualifier values. Any errors in this information are reported at this point. If there are no errors, SORT opens the input file and creates the output file. (Record interface does not use this step.) Next, SORT initializes key comparison information, generates the key comparison routine, and allocates the space needed for input and output buffers and the sort tree.

4.1.2 Sort Phase

After the initialization phase is complete, SORT begins reading records and placing them into a "sort tree" in the working set in memory. (For tag, address, or index sort only the key is placed in the sort tree.) This process continues until the sort tree is full or all records have been read. If all records fit into the sort tree, they are sorted in memory, and no work files are created.

If the working set becomes full, SORT begins to move records to a work file, where it builds sorted strings. The first record it removes from the sort tree is one with the smallest key.

SORT then reads the next input record into the sort tree. If there are more records to be read, it moves a second record from the sort tree to the string in the work file. This time it again selects a record with the smallest key, but not a key smaller than that of the record just moved. Moving a record with an equal or larger key creates a sorted string in the work file.

This process repeats until (1) all records have been read, or (2) SORT cannot transfer a record with an equal or higher key to the string. In the latter case, SORT begins building a second string, again selecting first a record with the smallest key from those in the sort tree.

When all records have been read into the sort tree, they are moved one by one in sorted order to the string being built in the work file. If at some point no record remaining in the sort tree has an equal or higher key than the record most recently moved into the work file, SORT begins a new string.

After all records have moved from the sort tree to the work file, the work file contains any number of strings, each of which is a set of sorted records (or keys, for other than record sort).

4.1.3 Merge Phase

In the internal merge phase SORT begins merging the sorted strings in the work file. Up to ten strings are merged into one ordered string. If the work file contains no more than ten strings, the records in the merged string are written to the output file or returned to the calling program. (In tag sort, a subroutine uses the merged string of keys to read the records in the sorted sequence before it writes them. In address sort the output file contains RFAs only; in index sort, it contains keys and RFAs.)

If the work file contains more than ten strings, the merged string is written to the second work file. Then another merge of up to ten strings takes place, and this merged string is also written to the second work file.

This internal merging continues until all sorted strings in the first work file have been merged and moved to the second work file. This process is one merge pass.

If the strings have been moved to work file #2, another merge pass begins, this time moving merged strings from work file #2 to work file #1. Again, if work file #2 contains no more than ten strings, the merged records are written directly to the output file or returned to the calling program.

The merging process continues until all records have been merged into one ordered string, which is the final sorted file.

4.1.4 Clean-Up Phase

After the last record is written, SORT closes the input and output files (for utility sort and file interface), closes and deletes the work files, and releases memory. Finally, for interactive or batch sorting operations, the utility gets the SORT statistics and prints or displays them, then returns to the VAX/VMS command interpreter.

4.2 Understanding and Using SORT Statistics

The /STATISTICS qualifier causes SORT to display statistics like those in the following example. You can use the SORT statistics to judge the efficiency of a sorting operation and determine the adjustments needed to improve performance.

```
Records read:    793                    Longest record length:  73
Records sorted: 793                     Input multi block count: 16
Records output: 793                     Output multi block count: 16
Maximum working set used: 200           Input multi buffer count: 2
Virtual memory added:    717312         Output multi buffer count: 2
Direct I/O count: 10                    Number of initial runs: 2
Buffered I/O count: 13                  Order of the merge:     2
Page faults:      713                   Number of merge passes: 1
Elapsed time:     00:00:06.51           CPU time: 331
```

If the number of **records read, sorted,** and **output** is not the same, this discrepancy indicates a problem with I/O or with the records themselves. The difference can be caused by input or output errors, null records in the file, or records containing invalid data in key fields. If fewer than ten records contain invalid data, SORT continues; if ten or more do, SORT stops executing.

Maximum working set used shows the size in blocks of the working set used for the sort. Adjusting this value is one of the ways to improve SORT efficiency; for medium and large files, the higher it is, the better.

The **multi block** and **buffer counts** indicate the amount of I/O optimization. Depending on the size of the working set, SORT creates for input and for output one or two buffers each. A buffer can hold from 1 to 16 blocks. Having two buffers permits overlap of reading and processing and thus speeds up the sort. The more blocks each buffer holds, the less time is spent in I/O. The closer these numbers are to the maximums of 2 and 16, the greater the optimization. If all counts are 1, no optimization took place.

The total of the two **I/O counts** is the number of I/O movements needed to get and write data. The lower these are, the more efficient the sort was. This count decreases as the multi block and buffer counts increase.

The **order of the merge** is the number of sorted strings that are merged in the final merge pass.

Initial runs are the sorted strings created in the work file. If all records were sorted in memory, this number is 0.

The **number of merge passes** is the number of times SORT must pass through a work file, merging the strings, until one sorted string results. The number of initial runs and the number of merge passes shows you how close the data is to fitting in memory. The higher these numbers are, particularly the number of passes, the longer SORT takes and the further away the working set size is from containing the data.

The number of **page faults** also indicates how well the data fit into memory. The higher the number of page faults, the less efficient the sort is.

Longest record length value is obtained from RMS unless the user supplies it. You can use it to check the RMS value.

Elapsed time is the total wall clock time in hours, minutes, seconds, and 1/100 seconds from the start to the end of the sort run. **CPU time** is the time in 1/100 seconds that the processor spends handling the data; it does not include I/O time or time waiting for another process to execute. The closer CPU time is to elapsed time, the better optimization you are seeing in I/O.

Virtual memory added is the number of bytes of virtual memory SORT used for the data.

4.3 What the User Can Do

Any software sorting routine can be written to work most efficiently in the environment in which it is likely to be used most frequently. Environment variables include such things as type of I/O devices, key data type, file size, and relative size of key and record. VAX–11 SORT/MERGE is designed for an environment of random access disk devices, fairly large files, and medium size records and keys.

SORT software automatically provides the most efficient sort for the data type(s) of your key(s). For each sort, it generates a key comparison routine tailored to the particular keys being sorted.

Two ways the user can improve SORT efficiency are by: (1) adjusting working set size and (2) placing work files effectively.

4.3.1 Working Set Quota

For SORT to work efficiently, it is important to use the largest working set possible. Normally, this means setting a working set limit to your authorized working set quota. For very small files, however, a smaller working set quota may be more efficient.

During the sort phase, records are read into the sort tree in memory (that is, the working set) until it is full. If the space in the working set cannot hold all the records, SORT transfers sorted strings to a work file on a temporary storage device. SORT must then take the time to perform merge passes until one sorted string results. If all records can be sorted in main memory, SORT does not need to perform any merge passes.

A larger working set also increases sort efficiency when the sort uses work files, as in most sorts of large files. SORT uses the larger working set space to create a larger sort tree. The larger the sort tree, the greater chance SORT has of creating longer and fewer sorted strings in the work file. Fewer strings means fewer merge passes and a faster sort.

4.3.2 Location of Work Files

Another important consideration is the location of the work files. Normally SORT places work files on your default device, but you can select a different device for any work file by specifying:

```
ASSIGN (device): SORTWORKn
```

The n represents the number of the work file (0 to 9).

You can increase SORT efficiency by placing work files on:

- The fastest device(s) available
- Device(s) having the least activity
- The least full device(s) available
- Separate devices

Placing work files on separate devices permits overlap of the read/write cycle. You can gain more speed by placing the input file and output file, as well as each work file, on different devices. The ideal configuration for SORT is to have the input and output file and each work file all on separate empty disks being used only by SORT during the sorting process. However, this is seldom possible, so the next best procedure is to place work files on devices meeting as many of these requirements as possible.

Usually there is no advantage to using more than two work files. However, if the available disks are too small or too full for SORT work files, increasing the number of work files makes each work file smaller so that it can fit onto a small or nearly full disk.

Because SORT does not create work files until it needs them, you do not gain in speed by specifying 0 work files to keep work space in real memory.

4.3.3 Process

Although typically you select a sort process for reasons other than efficiency, you should understand the differences in speed among the four sort processes. See Section 2.1.2 for a description of these differences. In any operation in which the sorted records will be retrieved in order, record sort is usually the fastest sort process.

If limited work space is a problem, consider using tag sort, which requires less space than record sort.

4.4 What the System Manager Can Do

The system manager can designate one batch queue for sorting jobs and give this queue a very large working set quota. He can also adjust parameter values for greatest SORT efficiency.

The system manager can determine the following SORT performance parameter values based on the overall system usage: number of users, sort process most commonly run, and the amount of real memory available.

- System per process working set quota (WSMAX)
- System per process virtual page count (VIRTUALPAGECNT)
- System per process section count (PROCSECTCNT)
- System modified page writer cluster factor (MPW_WRTCLUSTER)

The values recommended are based solely on SORT considerations; it is up to the system manager to integrate other system considerations with these in determining the final values.

4.4.1 Working Set Quota

The maximum for this value should be set to the largest size any sort job would ever need. For very large files, working sets of 500 to 1000 pages are not at all unreasonable, provided the system has enough physical memory to handle them. To prevent users from monopolizing real memory, the system manager can set individual maximums on a per user basis through the authorization file. The general rule is: the smaller the working set, the slower the sort.

4.4.2 Virtual Page Count

For this parameter the current value as well as the maximum value should be set to a minimum of 7 times the value of the working set quota maximum. When SORT begins executing it requests at least 7 times the working set quota of virtual memory from the system. If this parameter is too low, SORT may be unable to run.

4.4.3 Process Section Count

This parameter needs a value of 20 to guarantee that SORT will not fail. If the parameter is set too low, SORT cannot run in larger working sets because of internal mapping failures. SORT itself never needs a value higher than 20, but if a program that uses many process sections also calls SORT subroutines, more than 20 process sections may be required.

4.4.4 Modified Page Writer Cluster Factor

The value of this parameter never causes SORT to fail, but it can affect performance greatly. For any larger sorts (that is, using working sets of 250 pages or greater) the larger this parameter, the better. Values of 64 and up are not too large. Be sure to adjust MPW_HILIM and MPW_LOLIMIT accordingly. For more information refer to the SYSGEN procedures in the *VAX–11 Software Installation Guide.*

Appendix A
Error Conditions

You can encounter error conditions at three operating levels: with the VAX/VMS DCL command interpreter, with SORT/MERGE, and with VAX–11 RMS.

There are three categories of error conditions:

- Errors caused by I/O or other system failures
- Errors caused by misinformation passed to SORT as a parameter of a subroutine call
- Errors caused by invalid data in a key field

For the SORT/MERGE utility, all errors are signaled to the system; this signal causes a message to be displayed or printed. For the subroutines, an error status value is returned to the calling program.

SORT handles both fatal and warning errors. Fatal errors (severity level F) cause SORT to halt processing; warning errors (severity level W) cause a warning message to be displayed and allow sort processing to proceed.

Only invalid data errors and a few RMS errors cause warning error messages. System or I/O failures and bad subroutine parameters are fatal errors. For additional information regarding error condition handling, refer to the *VAX/VMS System Services Reference Manual.*

A.1 Command Interpreter Error Messages

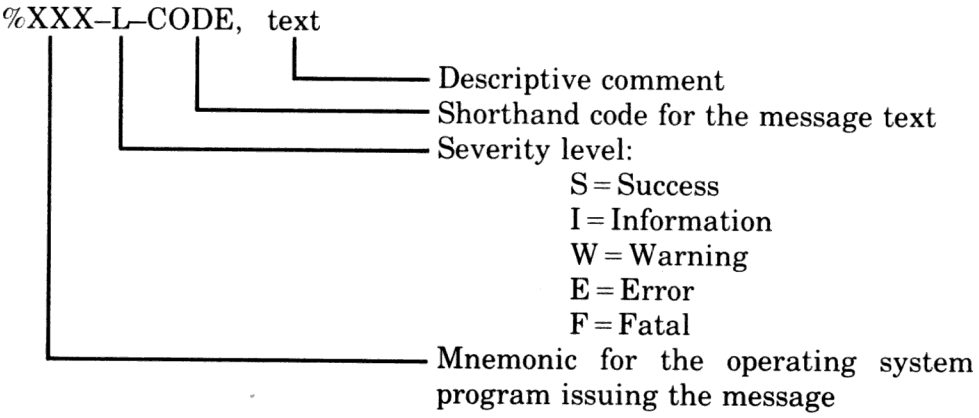
In interactive mode, when you enter a command line incorrectly, the command interpreter issues a descriptive error message telling you what was wrong. For example, if you specify more than one parameter for a command that accepts a single parameter, you receive the message:

```
%DCL-W-MAXPARAM, maximum parameter count exceeded
```

You must then retype the command line.

Other error messages may occur during execution of a command. These messages can indicate such errors as a nonexistent file or a conflict in qualifiers. Not all messages from the system indicate errors; other messages are informative or warn you of a particular condition.

The VAX/VMS system messages have the general format:



Example:

```
%SORT-W-CLOSEOUT, error closing output (output file-specification)
```

Because these messages are descriptive, you should be able to understand what you need to do differently when you issue the command again. But, if you do not, the *VAX/VMS System Messages and Recovery Procedures Manual* lists all the possible command interpreter error messages and describes what you can do to correct a command interpreter error.

A.2 SORT/MERGE Error Messages

The following VAX–11 SORT/MERGE error messages are listed in alphabetical order. They all have the same format as command interpreter messages, that is:

%SORT–(severity level)–(code),(text).

The error message descriptions observe the following conventions:

- Only the (code), (text) part of the message is shown in the following list.
- (filespec) indicates a file specification.
  For example: DB1:[153,10]TEST.TMP;3.
- (number) indicates the user entered numeric value.
- LRL means the longest record length (specified in bytes).

```
BAD_ADR, invalid descriptor address specified.
```

You passed the subroutine package an address for a descriptor, and the descriptor was invalid format. Character string descriptors in VAX consist of two longwords. The first word of the first longword contains the character string length in bytes. The second longword contains the address of the string.

User Action: Check character string format.

```
BADFIELD, (filespec, or field text that is invalid) field invalid at (number).
```

Bad data in key field or command. In this message, (number) indicates the record number of the record containing bad data in hex.

User Action: Check key field data type and the starting positions and lengths (see Section 2.1.2).

```
BAD_FILE, file size invalid.
```

You specified a negative file size or a zero file size. File size must be greater than zero.

User Action: Specify a file size greater than zero.

```
BAD_KEY, invalid key specification.
```

Either the key field size, position, data type, or order is incorrect within the key definition. Positions start at one and cannot be greater than the maximum record size. Size must be less than or equal to 255 for character data, 1, 2, 4, or 8 (byte, word, longword, or quadword) for binary data, and less than or equal to 31 for decimal.

User Action: See Section 2.1.2 and check the command string key specifications.

```
BAD_LEN, output record length less than 18 bytes for magtape.
```

Magnetic tape requires record lengths to be at least 18 bytes.

User Action: Check your output file record length.

```
BAD_LRL, input file (filespec).
Record size greater than specified LRL.
```

In reading the input file, SORT encountered a record longer than the specified LRL. The record will be truncated to the LRL and sorted.

User Action: Re-execute SORT with a larger LRL.

```
BAD_MERGE, merge order must be between 2 and 10.
```

You specified fewer than 2 or more than 10 input files to MERGE.

User Action: Specify between 2 and 10 input files.

```
BAD_ORDER, input file (number) is out of order.
```

A MERGE input file is not in order on the specified key. In this message (number) refers to the number of the input file in the list of input files. You must specify /CHECK_SEQUENCE to receive this message.

User Action: Sort the specified file and re-submit to MERGE.

```
BAD_ROUTIN, bad routine address has been passed.
```

You passed the subroutine package an incorrect address for a routine.

User Action: See Chapter 3 and specify the correct routine address.

```
BAD_SPEC, invalid specification file record.
FIELD:(record specification).
```

An incorrect field was specified in the specification file record. (record specification) indicates bad record contents.

User Action: See Section 2.4 for descriptions of specification file record formats and change field specifications.

```
BAD_TYPE, invalid sort process.
```

You passed the subroutine package a sort type code of less than 1 or greater than 4 if file I/O or not equal to 1 if record I/O, or an invalid key word in command /PROCESS. Legal values are 1–4 for file I/O, and RECORD, TAG, INDEX, or ADDRESS for command /PROCESS parameter.

User Action: Specify a different sorting process.

```
CLEAN_UP, failed to reinitialize work area and files.
```

SORT was unable to deallocate the extra virtual memory, deassign work file channels, or readjust working set size. For the SORT utility, this is a warning of little importance. For the SORT subroutine packages, this could mean a failure to be able to recall SORT from the same program until it has exited. This is an internal error.

User Action: Exit from the user program before re-executing SORT.

CLOSEIN, error closing (filespec) as input.

An error occurred closing an input file. This message is usually accompanied by an RMS message indicating the reason for the failure.

User Action: Take corrective action based on the associated message.

CLOSEOUT, error closing (filespec) as output.

An error occurred closing an output file. This message is usually accompanied by an RMS message indicating the reason for the failure.

User Action: Take corrective action based on the accompanying message.

EXTEND, failed to extend work file.

SORT failed to extend a user's temporary work file. Either the device is full, or the user does not have extend privilege.

User Action: See Section 4.3.2 and reassign work files to a different device with more space, and make sure you have extend privilege on that directory.

INCONSIS, inconsistent data in file (filespec).

If you specified /OVERLAY plus other output file qualifiers, SORT will verify that the information in the existing file matches the information you provided. If it does not, this error message is reported. Unless you specifically want a verification, /OVERLAY should be used without other qualifiers.

User Action: Check the command string output file qualifiers (see Section 2.1.3).

IND_OVR, indexed sequential output requires overlay qualifier.

You specified indexed output file organization without specifying /OVERLAY.

User Action: You must create the indexed file first with a utility such as the RMS DEFINE utility. The primary key of the file should be the same as the sort key for efficiency but is not required to be. Then you must specify /OVERLAY in the SORT command string.

INP_FILES, too many input files.

You listed more than 10 input files.

User Action: Reduce the number of input files or combine them so that you list no more than ten.

KEY_LEN, Key length invalid, Key number (number), size (number).

The key size is incorrect for the data type, or the total key size is greater than 255 (or 251 if you are using /STABLE).

User Action: See Section 2.1.2 and specify correct key field size. Size must be less than or equal to 255 for character data, 1, 2, or 4 for binary data, and less than or equal to 31 for decimal. Also, only ascending or descending order is allowed.

LRL_MISS, LRL must be specified.

If record I/O interface subroutine package is selected, the longest record length (LRL) must be passed to SORT or MERGE in the call.

User Action: Specify LRL.

MAP, failed to map work file.

This is an internal SORT failure.

User Action: Verify that the system parameter "maximum process sections" has been set up at 20. If it has, then report this failure to a specialist. Otherwise, set that system parameter to 20.

MISS_KEY, Key specification missing.

SORT did not find any key definition in either the command line or specification file, or in the parameters to the subroutine package.

User Action: You must input at least one key definition in one of these three areas.

NO_WRK, need work files cannot do SORT in memory.

You specified /WORK_FILES=0 indicating the data would fit in memory, but the data was too large.

User Action: Either increase the working set quota, or allow SORT to use two or more work files.

NUM_KEY, too many keys specified.

Up to ten key definitions are allowed. Either too many have been specified, or the NUMBER value is wrong.

User Action: See Section 2.1.2 and check your command string key field specifications.

OPENIN, error opening (filespec) as input.

An input file cannot be opened. This message is usually accompanied by an RMS message indicating the reason for the failure.

User Action: Take corrective action based on the associated message.

OPENOUT, error opening (filespec) as output.

An output file cannot be opened. This message is usually accompanied by an RMS message indicating the reason for failure.

User Action: Take corrective action based on the associated message.

READERR, error reading (filespec).

An input file record specified cannot be read. This message is usually accompanied by an RMS message indicating the reason for the failure.

User Action: Take corrective action based on the associated message.

SORT_ON, sort already in progress.

You tried to call the SORT subroutine package with calls in the wrong order, or to recall it before it finished running the previous sort.

User Action: Reorder the subroutine calls and then re-execute SORT.

TOO_BIG, record size larger than declared maximum.

A record is larger than the LRL you specified.

User Action: Change LRL value.

VAR_FIX, cannot change variable length records into fixed length.

You specified variable length input records and requested fixed length output.

User Action: Output records must be variable or controlled in this case.

VM_FAIL, failed to get required virtual memory (number).

SORT could not get the amount of virtual memory required for the sort. (number) indicates the number of bytes needed.

User Action: If the SORT utility is being run, decrease the working set quota; if either SORT subroutine package is being run, either decrease the quota or return some memory to the system inside the user's program before calling SORT.

```
WORK_DEV, work file (filespec) device specified not random
access or not local.
```

Work files must be specified for random access devices that are local to the CPU the sort is being performed on (that is, not on node in a network). Random access devices are disk devices.

User Action: Specify the correct device.

```
WRITEERR, error writing (filespec).
```

An output file record cannot be written. This message is usually accompanied by an RMS message indicating the reason for the failure.

User Action: Take corrective action based on the associated message.

```
WS_FAIL, failed to get required working set space (number).
```

SORT could not get the required amount of real memory space. A minimum 75 page working set is needed. (number) indicates number of pages available.

User Action: Increase the working set quota.

A.3 VAX–11 RMS Error Codes

If an error occurs in an input or output operation, an RMS code is returned to your program by the operating system. Often RMS messages are linked with SORT/MERGE messages.

All VAX–11 RMS error messages have the same format as command interpreter messages, that is:

%RMS–(severity level)–(code),(text).

Like the command interpreter error messages, RMS error messages are descriptive. The descriptions should make clear what you need to do differently. Refer to the *VAX–11 Record Management Services Reference Manual* for a list of RMS error codes and recovery procedures.

Appendix B
Octal/Hexadecimal/Decimal Conversion

B.1 Octal/Decimal Conversion

To convert a number from octal to decimal, locate in each column of the table the decimal equivalent for the octal digit in that position. Add the decimal equivalents to obtain the decimal number.

To convert a decimal number to octal:

1. Locate the largest decimal value in the table that will fit into the decimal number to be converted
2. Note its octal equivalent and column position
3. Find the decimal remainder

Repeat the process on each remainder. When the remainder is 0, all digits will have been generated.

	8^5	8^4	8^3	8^2	8^1	8^0
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	1
2	65,536	8,192	1,024	128	16	2
3	98,304	12,228	1,536	192	24	3
4	31,072	16,384	2,048	256	32	4
5	163,840	20,480	2,560	320	40	5
6	169,608	24,576	3,072	384	48	6
7	229,376	28,672	3,584	448	56	7

B.2 Powers of 2 and 16

Powers of 2

2**n	n
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20
2094304	21
4194304	22
8388608	23
16777216	24

Powers of 16

16**n	n
1	0
16	1
256	2
4096	3
65536	4
1048576	5
16777216	6
268435456	7
4294967296	8
68719476736	9
1099511627776	10
17592186044416	11
281474976710656	12
4503599627370496	13
72057594037927936	14
1152921504606846976	15

B.3 Hexadecimal to Decimal Conversion

For each integer position of the hexadecimal value, locate the corresponding column integer and record its decimal equivalent in the conversion table A.5. Add the decimal equivalent to obtain the decimal value.

Example:

```
D0500AD0 (16)   =       ?(10)

D0000000        =  3,489,660,928
  500000        =      5,242,880
     A00        =          2,560
      D0        =            208

D0500AD0        =  3,494,904,576
```

B.4 Decimal to Hexadecimal Conversion

1. Locate in the conversion table A.5 the largest decimal value that does not exceed the decimal number to be converted.
2. Record the hexadecimal equivalent followed by the number of zeros (0) that corresponds to the integer column minus one.
3. Subtract the table decimal value from the decimal number to be converted.
4. Repeat steps 1–3 until the subtraction balance equals zero (0). Add the hexadecimal equivalents to obtain the hexadecimal value.

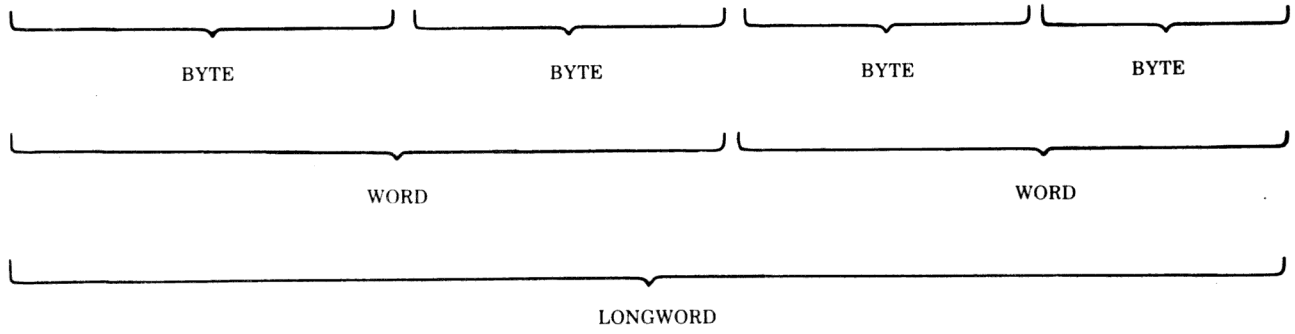
Example:

```
22,466 (10)      =  ?(16)

20,480           =  5000        22,466
 1,792           =   700       -20,480
   192           =    C0       --------
     2           =     2         1,986
------           =  ----      - 1,792
22,466           =  57C2       --------
                                    194
                               -    192
                               --------
                                      2
                               -      2
                               --------
                                      0
```

B.5 Hexadecimal Integer Columns

8		7		6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,643	8	134,217,728	8	8,338,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,916	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15



Appendix C
The ASCII Character Set Collating Sequence

ASCII Character	Hexadecimal Number	ASCII 8–Bit Octal	Decimal	ASCII Character	Hexadecimal Number	ASCII 8–Bit Octal	Decimal
NUL	00	000	0	FS	1C	034	28
SOH	01	001	1	GS	1D	035	29
STX	02	002	2	RS	1E	036	30
ETX	03	003	3	US	1F	037	31
EOT	04	004	4	SP	20	040	32
ENQ	05	005	5	!	21	041	33
ACK	06	006	6	"	22	042	34
BEL	07	007	7	#	23	043	35
BS	08	010	8	$	24	044	36
HT	09	011	9	%	25	045	37
LF	0A	012	10	&	26	046	38
VT	0B	013	11	'	27	047	39
FF	0C	014	12	(	28	050	40
CR	0D	015	13	)	29	051	41
SO	0E	016	14	*	2A	052	42
SI	0F	017	15	+	2B	053	43
DLE	10	020	16	,	2C	054	44
DC1	11	021	17	–	2D	055	45
DC2	12	022	18	.	2E	056	46
DC3	13	023	29	/	2F	057	47
DC4	14	024	20	0	30	060	48
NAK	15	025	21	1	31	061	49
SYN	16	026	22	2	32	062	50
ETB	17	027	23	3	33	063	51
CAN	18	030	24	4	34	064	52
EM	19	031	25	5	35	065	53
SUB	1A	032	26	6	36	066	54
ESC	1B	033	27	7	37	067	55

(continued on next page)

ASCII Character	Hexadecimal Number	ASCII 8–Bit Octal	Decimal	ASCII Character	Hexadecimal Number	ASCII 8–Bit Octal	Decimal
8	38	070	56	\	5C	134	92
9	39	071	57	]	5D	135	93
:	3A	072	58	^	5E	136	94
;	3B	073	59	_	5F	137	95
<	3C	074	60	`	60	140	96
=	3D	075	61	a	61	141	97
>	3E	076	62	b	62	142	98
?	3F	077	63	c	63	143	99
@	40	100	64	d	64	144	100
A	41	101	65	e	65	145	101
B	42	102	66	f	66	146	102
C	43	103	67	g	67	147	103
D	44	104	68	h	68	150	104
E	45	105	69	i	69	151	105
F	46	106	70	j	6A	152	106
G	47	107	71	k	6B	153	107
H	48	110	72	l	6C	154	108
I	49	111	73	m	6D	155	109
J	4A	112	74	n	6E	156	110
K	4B	113	75	o	6F	157	111
L	4C	114	76	p	70	160	112
M	4D	115	77	q	71	161	113
N	4E	116	78	r	72	162	114
O	4F	117	79	s	73	163	115
P	50	120	80	t	74	164	116
Q	51	121	81	u	75	165	117
R	52	122	82	v	76	166	118
S	53	123	83	w	77	167	119
T	54	124	84	x	78	170	120
U	55	125	85	y	79	171	121
V	56	126	86	z	7A	172	122
W	57	127	87	{	7B	173	123
X	58	130	88	|	7C	174	124
Y	59	131	89	}	7D	175	125
Z	5A	132	90	~	7E	176	126
[	5B	133	91	DEL	7F	177	127

Appendix D
Data Types

The data type refers to the way bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand.

VAX–11 SORT/MERGE uses the following data types

- Integer and Floating Point Data Types

  Binary (1, 2, 4, and 8–byte)
  F_Floating
  D_Floating
  G_Floating
  H_Floating

- Character String Data Type

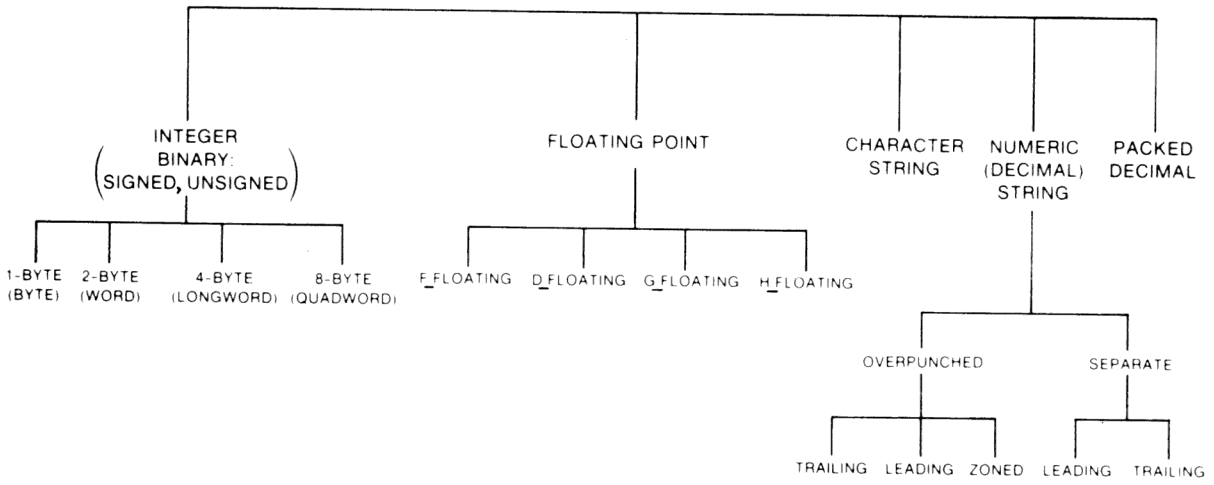
- Numeric String Data Types

  Trailing Overpunched Decimal String
  Leading Overpunched Decimal String
  Zoned Decimal String
  Leading Separate Decimal String
  Trailing Separate Decimal String

- Packed Decimal String Data Type

Figure D–1 summarizes the relationship of these data types.

Figure D–1: VAX–11 SORT/MERGE Data Types



D.1 Integer and Floating Point Data Types

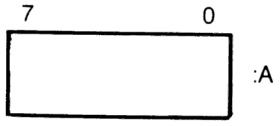
In the following discussion of integer and floating data types, the address of the datum in memory is the address of the byte of the datum with the lowest address. When depicted, this lowest byte is shown on the right and in discussions this is what is meant when the word "right" is used.

D.1.1 Integer Data (Binary)

VAX–11 SORT/MERGE uses integer data types of 8–, 16–, 32–, and 64–bit sizes. They are termed byte, word, longword, and quadword integers respectively. The integer data types are stored in memory in a binary format and can be treated as either signed or unsigned quantities. In the case of signed quantities, the integer is represented in two's complement form. This means that positive numbers have a zero most significant bit (MSB) and the representation of a negative number is one greater than the bit-by-bit complement of its positive counterpart. Thus the MSB is always zero for positive values and one for negative values. When treated as unsigned quantities, integers extend upward from 0.

One-Byte Binary (Byte)

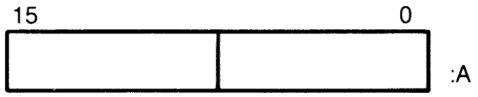
A byte is eight contiguous bits starting on an addressable byte boundary. The bits are numbered from the right 0 through 7:



A byte is specified by its address A. When interpreted as a signed quantity, a byte is a two's complement integer with bits increasing in significance from 0 through 6, and with bit 7 designating the sign. The value of the integer is in the range –128 through 127. For the purposes of addition, subtraction, and comparison, VAX–11 instructions also provide direct support for the interpretation of a byte as an unsigned integer with bits increasing in significance from 0 through 7. The value of the unsigned integer is in the range 0 through 255.

Two-Byte Binary (Word)

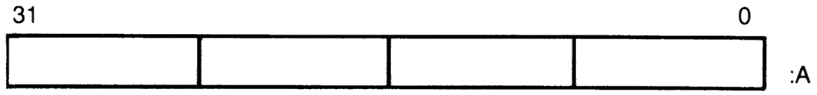
A word is two contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 15:



A word is specified by its address A, the address of the byte containing bit 0. When interpreted as a signed quantity, a word is a two's complement integer with bits increasing in significance from 0 through 14, and with bit 15 designating the sign. The value of the integer is in the range –32,768 through 32,767. For the purposes of addition, subtraction, and comparison, VAX–11 instructions also provide direct support for the interpretation of a word as an unsigned integer with bits increasing in significance from 0 through 15. The value of the unsigned integer is in the range 0 through 65,535.

Four-Byte Binary (Longword)

A longword is four contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 31:

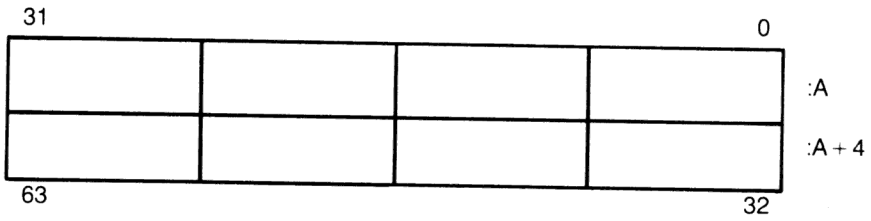


A longword is specified by its address A, the address of the byte containing bit 0. When interpreted as a signed integer, a longword is a two's complement integer with bits increasing in significance from 0 through 30, and with bit 31 designating the sign. The value of the integer is in the range –2,147,483,648 through 2,147,483,647. For addition, subtraction, and comparison, VAX–11 instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits increasing in significance from 0 through 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

Note that the longword format is different from the longword format defined by the PDP–11 FP–11. In that format, bits increase in significance from 16 through 31 and 0 through 14. Bit 15 is the sign bit. Most DIGITAL software, and in particular PDP–11 FORTRAN and COBOL, use the VAX–11 longword format.

Eight-Byte Binary (Quadword)

A quadword is eight contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 63:



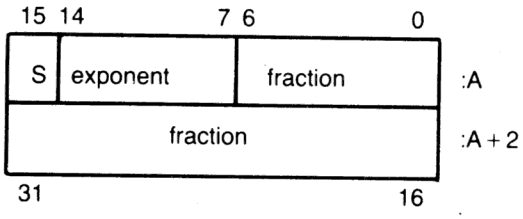
A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted as a signed integer, a quadword is a two's complement integer with bits increasing in significance from 0 through 62, and with bit 63 designating the sign. The value of the integer is in the range –2**63 to 2**63–1.

D.1.2 Floating Point Data

The floating point data types are used to represent approximations to quantities using a scientific notation consisting of a sign, the exponent of a power of two, and a fraction between .5 (inclusive) and 1.0 (exclusive). The value of a floating point number is the sign applied to the fractional part multiplied by two raised to the power specified by the exponent part. VAX–11 SORT/MERGE uses floating point data types of 32–, 64–, and 128–bit sizes.

F_Floating

An F_floating datum is four contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 31.

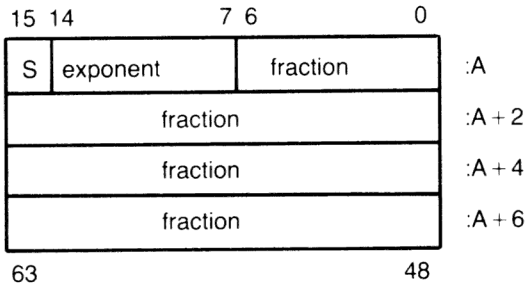


An F_floating datum is specified by its address A, the address of the byte containing bit 0. The form of an F_floating datum is sign magnitude with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24–bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 16 through 31 and 0 through 6. The 8–bit exponent field encodes the values 0 through 255. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the F_floating datum has a value of 0. Exponent values of 1 through 255 indicate true binary exponents of –127 through +127. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The magnitude of an F_floating datum

is in the approximate range .29*10**–38 through 1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, that is, typically 7 decimal digits.

D_Floating

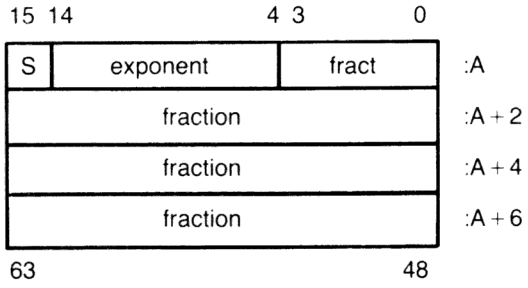
The D_floating datum is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 63:



A D_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a D_floating datum is identical to the F_floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits increase in significance from 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions and approximate range of values are the same for both D_floating and F_floating. The precision of a D_floating datum is approximately one part in 2**55, that is, typically 16 decimal digits.

G_Floating

A G_floating datum is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 63:

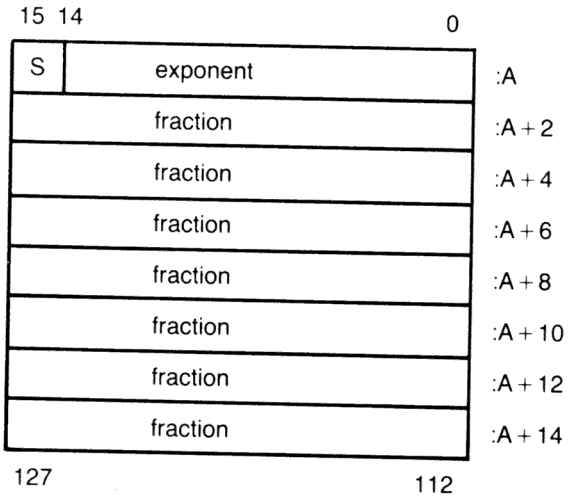


A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits 14:4 an excess 1024 binary exponent, and bits 3:0 and 63:16 a normalized 53–bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11–bit exponent field encodes the values 0 through 2047. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0. Exponent values of 1 through 2047

indicate true binary exponents of –1023 through +1023. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of a G_floating datum is in the approximate range .56*10**–308 through .9*10**308. The precision of a G_floating datum is approximately one part in 2**52, that is, typically 15 decimal digits.

H_Floating

An H_floating datum is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labelled from the right 0 through 127:

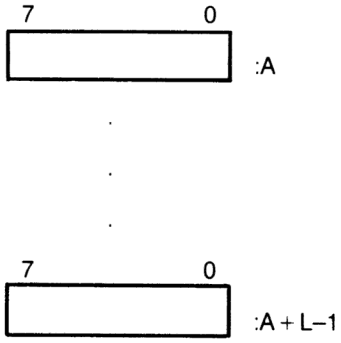


An H_floating datum is specified by its address A, the address of the byte containing bit 0. The form of an H_floating datum is sign magnitude with bit 15 the sign bit, bits 14:0 an excess 16384 binary exponent, and bits 127:16 a normalized 113–bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31. The 15–bit exponent field encodes the values 0 through 32767. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the H_floating datum has a value of 0. Exponent values of 1 through 32767 indicate true binary exponents of –16383 through +16383. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of an H_floating datum is in the approximate range .84*10**–4932 through .59*10**4932.. The precision of an H_floating datum is approximately one part in 2**112, that is, typically 33 decimal digits.

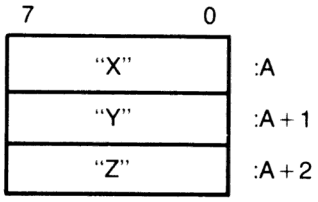
D.2 Character String Data Type

The character string is a data type used to represent strings of characters such as names, data records, or text.

A character string is a contiguous sequence of bytes in memory. A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. Thus the format of a character string is:



The address of a string specifies the first character of a string. For example, "XYZ" is represented:



The length L of a string is in the range 0 through 65,535.

D.3 Numeric String (Decimal) Data Types

The numeric string data types are used to represent fixed scaled quantities in a form close to their external representation. For programs that are input/output intensive rather than computation intensive, this presentation is frequently more efficient. The decimal form also provides greater precision than floating point and greater range than integer. There are two forms of decimal data on VAX–11: (1) the decimal string data types in which each decimal digit occupies one byte, and (2) a more compact form (discussed in the next section) in which two decimal digits are packed into one byte. These are termed numeric and packed decimal strings respectively. Because the numeric string form must represent many external data arrangements exactly, it can be specified in several forms. Two significant distinguishing characteristics are: (1) whether the sign appears before the first digit or after the last digit, and (2) whether the sign is superimposed on the digit or separated from it.

D.3.1 Overpunched Decimal Strings (Trailing, Zoned, Leading)

A decimal string is a contiguous sequence of bytes in memory. The string is specified by two attributes: the address A of the first byte (most significant digit) of the string, and the length L of the string in bytes.

All bytes of a zoned or trailing overpunched decimal string, except the least significant digit byte, must contain an ASCII decimal digit character (0–9). For a leading overpunched numeric string, all bytes except the most significant digit byte must contain an ASCII decimal digit character (0–9). The representation for these digits is:

digit	decimal	hex	ASCII character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The remaining byte represents an encoding of both the remaining digit and the sign of the decimal string. Thus, for zoned or trailing overpunched, the highest addressed byte represents an encoding of both the least significant digit and the sign of the numeric string. For leading overpunched, the lowest addressed byte represents an encoding of both the most significant digit and the sign of the numeric string. The VAX–11 numeric string instructions support any encoding; however, there are preferred encodings used by DIGITAL software. These include (1) zoned numeric, (2) trailing overpunched numeric, and (3) leading separate numeric. Because the overpunch format has been used by compilers of many manufacturers over many years, and because various card encodings are used, several variations in overpunch format have evolved. Typically, these alternate forms are accepted on input. The valid representations of the digit and sign in these formats is:

	Zoned Numeric Format			Overpunch Format			
						ASCII char.	
digit	decimal	hex	ASCII char.	decimal	hex	norm	alt.
0	48	30	0	123	7B	{	0|?
1	49	31	1	65	41	A	1
2	50	32	2	66	42	B	2
3	51	33	3	67	43	C	3
4	52	34	4	68	44	D	4
5	53	35	5	69	45	E	5
6	54	36	6	70	46	F	6
7	55	37	7	71	47	G	7
8	56	38	8	72	48	H	8
9	57	39	9	73	49	I	9
–0	112	70	p	125	7D	}	]:!
–1	113	71	q	74	4A	J	
–2	114	72	r	75	4B	K	
–3	115	73	s	76	4C	L	
–4	116	74	t	77	4D	M	
–5	117	75	.u	78	4E	N	
–6	118	76	v	79	4F	O	
–7	119	77	w	80	50	P	
–8	120	78	x	81	51	Q	
–9	121	79	y	82	52	R	

The length L of an overpunched numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a 0 length string is identically 0.

The address A of the string specifies the byte of the string containing the most significant digit. Digits of decreasing significance are assigned to increasing addresses. Thus "123" is represented:

Zoned Format

7 4	3 0	
3	1	:A
3	2	:A+1
3	3	:A+2

Trailing Overpunch Format

7 4	3 0	
3	1	:A
3	2	:A+1
4	3	:A+2

Leading Overpunch Format

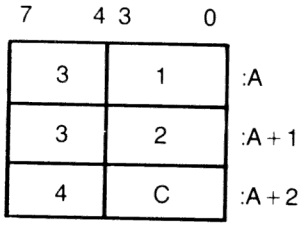
7 4	3 0	
4	1	:A
3	2	:A+1
3	3	:A+2

And "–123" is represented:

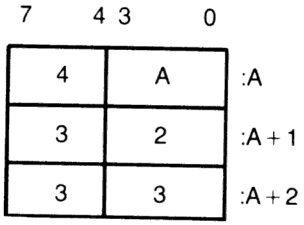
Zoned Format

7 4	3 0	
3	1	:A
3	2	:A+1
7	3	:A+2

Trailing Overpunch Format



Leading Overpunch Format



D.3.2 Leading Separate and Trailing Separate Decimal Strings

A decimal string is a contiguous sequence of bytes in memory. The string is specified by two attributes: the address A of the first byte of the string, and a length L that is the length of the string in digits and NOT the length of the string in bytes. In a leading separate numeric string, the first byte contains the sign character; in a trailing separate number string, the last byte contains the sign character. The number of bytes in a leading or trailing separate numeric string is L+1.

The sign of a leading separate or trailing separate decimal string is stored in a separate byte. Valid sign bytes are:

sign	decimal	hex	ASCII character
+	43	2B	+
+	32	20	<blank>
−	45	2D	−

The preferred representation for "+" is ASCII "+". All subsequent bytes contain an ASCII digit character:

digit	decimal	hex	ASCII character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The length L of a leading or trailing separate numeric string must be in the range 0 to 31 (0 to 31 digits). The value of a 0 length string is identically 0.

The address A of the leading separate decimal string specifies the byte of the string containing the sign. The address A of the trailing separate decimal string specifies the byte of the string containing the high order digit. Digits of decreasing significance are assigned to bytes of increasing addresses. Thus "+123" is:

Leading Separate Format

7 4	3 0	
2	B	:A
3	1	:A+1
3	2	:A+2
3	3	:A+3

Trailing Separate Format

7 4	3 0	
3	1	:A
3	2	:A+1
3	3	:A+2
2	B	:A+3

And "–123" is:

Leading Separate Format

7 4	3 0	
2	D	:A
3	1	:A+1
3	2	:A+2
3	3	:A+3

Trailing Separate Format

7 4	3 0	
3	1	:A
3	2	:A+1
3	3	:A+2
2	D	:A+3

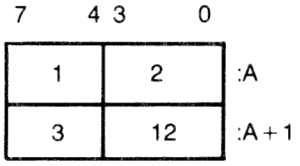
D.4 Packed Decimal String

A packed decimal string is a contiguous sequence of bytes in memory. A packed decimal string is specified by two attributes: the address A of the first byte of the string and a length L that is the number of digits in the string and NOT the length of the string in bytes. The bytes of a packed decimal string are divided into two 4–bit fields (nibbles) that must contain decimal digits except the low nibble (bits 3:0) of the last (highest addressed) byte, which must contain a sign. The representation for the digits and sign is:

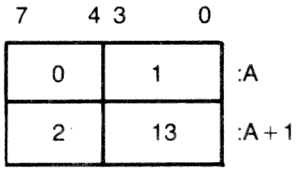
digit or sign	decimal	hex
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10, 12, 14, or 15	A, C, E, or F
–	11 or 13	B or D

The preferred sign representation is 12 for "+" and 13 for "–". The length L is the number of digits in the packed decimal string (not counting the sign) and must be in the range 0 through 31. When the number of digits is odd, the digits and the sign fit in L/2 (integer part only)+1 bytes. When the number of digits is even, it is required that an extra "0" digit appear in the high nibble (bits 7:4) of the first byte of the string. Again, the length in bytes of the string is L/2+1.

The address A of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. Thus "+123" has length 3 and is represented:



And "–12" has length 2 and is represented:



Glossary

Address

A name, label or number that identifies a location in memory where data is stored.

ASCII

Acronym for American Standard Code for Information Interchange; a set of 128 eight-bit characters used for standard representation of data.

Batch Processing

A mode of processing in which all commands to be executed by the operating system (and, optionally, data to be used as input to the commands) are placed in a file or punched onto cards and submitted to the system for execution.

Bit Flag

See **Flag.**

Block

The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices).

Bucket

A storage structure of 1 to 32 blocks used to store and transfer blocks of data in files with a relative file organization.

Buffer

An internal memory area used for temporary storage of data during input or output operations.

Byte

The smallest addressable unit of information; eight contiguous bits that are operated on by the computer as a unit.

Call

To transfer control to a specified routine.

Character

A single printable letter, number, or symbol represented by one byte.

Collating Sequence

An order assigned to the characters of a character set (for example, ASCII and EBCDIC) used for sequencing purposes.

Command

An instruction, generally an English word, typed by the user at a terminal or included in a command file. A command requests the software monitoring a terminal or reading a command file to perform some pre-defined activity. For example, typing the SORT command requests the system to invoke the SORT utility. An entire command can consist of the command name, qualifiers, and parameters.

Command Procedure

A file containing a sequence of commands to be executed by the operating system. The command procedure can be submitted for execution at the terminal or as a batch job.

Contiguous Blocks

Physically adjacent and/or consecutively numbered blocks of data.

Control Bytes

See **Variable with Fixed-Length Control (VFC) Record Format.**

Data Type

An interpretation applied to a series of bits, based on the way in which the bits are grouped. The data type of an operand identifies its size and the significance of the bits in the operand.

Default

An assumed value supplied by the system when the user does not specify a command qualifier.

Descriptor

A quadword data structure used in calling sequences for passing character strings. The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

Digit

One symbol in a numbering system.

EBCDIC

Acronym for Extended Binary Coded Decimal Interchange Code; a set of eight-bit characters used for representing data.

Field

A logically distinguishable byte or group of bytes within a record, usually a unit of information.

File

An organized collection of records maintained in an accessible storage area, such as disk or tape. The user can manipulate a file as a unit.

File Organization

The particular VAX–11 RMS file structure used to arrange the data constituting a file on a mass storage medium. See **Sequential**, **Relative**, and **Indexed-Sequential File Organizations**.

File Specification

A unique name for a file on a mass storage medium. It identifies the node, the device, the directory name, the file name, and the version number under which a file is stored.

Fixed-Length Record Format

A file format in which all records have the same length.

Flag

A bit that can be set to invoke the execution of some sequence of instructions; frequently, an indicator used to tell some later part of a program that a certain condition occurred earlier.

Floating Point Data

A method for compact storage of very large and very small numbers. The number is represented in two parts, a fraction and an exponent.

Indexed-Sequential File Organization

A file organization in which records are arranged according to a primary key specified by the user. In addition to the records, the file contains the primary key index and (optionally) one or more alternate key indexes. RMS places the records in order by the primary key index and maintains this sequence when new records are added. Records can be accessed sequentially by index key or randomly by alternate index key.

Input File

The file containing the records you wish to sort or merge.

Key

The data field in a record that contains the information by which the user wants to arrange the records.

Longword

Four contiguous bytes (32 bits) starting on an addressable byte boundary.

Merge

A process which combines records into one sequenced file from two or more similarly sequenced files.

Output File

The file created by running SORT or MERGE.

Packed Decimal

A method for compact storage of a decimal number. Two digits are stored in each 8–bit byte, taking advantage of the fact that only 4 bits are required to represent the numbers 0 through 9. The sign resides in the last byte of the low-order digit.

Parameter

Object of a command. A parameter can be a file specification, a keyword option, or a symbol value.

Qualifier

A portion of a command string that modifies a command verb or command parameter. A qualifier follows the command verb or parameter to which it applies and is in the format: /qualifier[=option].

Record

A set of related data treated as a unit; usually, one unit in a file.

Record Format

The way a record physically appears on the recording surface of the storage medium. See **Fixed-Length**, **Variable-Length**, and **Variable with Fixed-Length Control Record Formats**.

Record's File Address (RFA)

The unique address of a record in a disk file that allows records to be accessed randomly regardless of file organization.

Relative File Organization

A file organization in which the file contains fixed length record cells. Each cell is assigned a consecutive number representing its position relative to the beginning of the file. Records within each cell can be as big as or smaller than the cell. Relative file organization permits sequential record access, random record access by record number, and random record access by record's file address.

RMS

VAX–11 Record Management Services; the file and record access subsystem of the VAX/VMS operating system. VAX–11 RMS is a set of software routines in the operating system that maintains, organizes, and processes data, and allows access to files and records.

Routine

A set of sequenced instructions that causes a computer to perform particular functions.

Sequential File Organization

A file organization in which records are arranged in the sequence they were originally written. The records can be fixed length or variable length. Sequential file organization permits sequential record access and random access by record's file address. Sequential file organization with fixed length records also permits random access by relative record number.

Sort

A process that arranges records in a prescribed sequence.

Specification File

A file of records that contain the SORT command qualifiers in prescribed positions. A specification file must include one Header Record and as many Field Specification Records as there are keys in the sort.

Status Code

A longword value that indicates the success or failure of a specific function.

Subroutine

A subsidiary routine that executes when called by another program. A subroutine is often called repeatedly until a certain condition is met.

Utility

A program that provides a set of related general purpose functions.

Variable-Length Record Format

A file format in which records are not necessarily the same length.

Variable with Fixed-Length Control (VFC) Record Format

A record format in which a fixed-length control area is prefixed to a variable-length record. This control area contains information about the record that may have no bearing on the contents of the record, for example, file line numbers and/or print format controls.

Volume

A mass storage medium such as a disk pack or reel of magnetic tape.

Word

Two contiguous bytes (16 bits) starting on an addressable byte boundary.

Work File

A collection of sorted records created during the sort operation and released when the sort is finished.

Zoned Numeric Format

A specific ASCII coded decimal data type where the number sign and the least significant digit are combined into a single hexadecimal code.

Index

R

S

T

U

V

W

Z

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**

In Canada call **800–267–6146**

In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed with the Local Digital Subsidiary:
809–754–7575

Reader's Comments

Note: This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

--- Do Not Tear - Fold Here and Tape ---

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK1-2/ H3
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK N.H. 03054

--- Do Not Tear - Fold Here and Tape ---